DB2 11 for z/OS

SQL Reference

SC19-4066-00

DB2 11 for z/OS

SQL Reference

SC19-4066-00

Note
Before using this information and the product it supports, be sure to read the general information under "Notices" at the end of this information.

First edition (October 2013)

This edition applies to DB2 11 for z/OS (product number 5615-DB2), DB2 11 for z/OS Value Unit Edition (product number 5697-P43), and to any subsequent releases until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Specific changes are indicated by a vertical bar to the left of a change. A vertical bar to the left of a figure caption indicates that the figure has changed. Editorial changes that have no technical significance are not noted.

Contents

About this information

This book is a reference for Structured Query Language (SQL) for DB2 Universal Database™ for z/OS®. Unless otherwise stated, references to SQL in this book imply SQL for DB2® UDB for z/OS, and all objects described in this book are objects of DB2 UDB for z/OS.

This information assumes that your DB2 subsystem is running in Version 11 new-function mode. Generally, new functions that are described, including changes to existing functions, statements, and limits, are available only in new-function mode, unless explicitly stated otherwise. Exceptions to this general statement include optimization and virtual storage enhancements, which are also available in conversion mode unless stated otherwise.

The syntax and semantics of most SQL statements are essentially the same in all IBM® relational database products, and the language elements common to the products provide a base for the definition of IBM SQL. Consult IBM DB2 Universal Database SQL Reference for Cross-Platform Development if you intend to develop applications that adhere to IBM SQL.

Who should read this information

This information is intended for end users, application programmers, system and database administrators, and for persons involved in error detection and diagnosis.

This information is a reference rather than a tutorial. It assumes that you are already familiar with SQL programming concepts.

When you first use this information, consider reading Chapters 1 and 2 sequentially. These chapters describe the basic concepts of relational databases and SQL, the basic syntax of SQL, and the language elements that are common to many SQL statements. The rest of the chapters and appendixes are designed for the quick location of answers to specific SQL questions. They provide you with query forms, SQL statements, SQL procedure statements, DB2 limits, SQLCA, SQLDA, catalog tables, and SQL reserved words.

DB2 Utilities Suite

Important: In this version of DB2 for z/OS, the DB2 Utilities Suite is available as an optional product. You must separately order and purchase a license to such utilities, and discussion of those utility functions in this publication is not intended to otherwise imply that you have a license to them.

The DB2 Utilities Suite can work with DB2 Sort and the DFSORT program. You are licensed to use DFSORT in support of the DB2 utilities even if you do not otherwise license DFSORT for general use. If your primary sort product is not DFSORT, consider the following informational APARs mandatory reading:

- II14047/II14213: USE OF DFSORT BY DB2 UTILITIES
- II13495: HOW DFSORT TAKES ADVANTAGE OF 64-BIT REAL ARCHITECTURE

These informational APARs are periodically updated.

Related information
DB2 utilities packaging (Utility Guide)

Terminology and citations

When referring to a DB2 product other than DB2 for z/OS, this information uses the product's full name to avoid ambiguity.

The following terms are used as indicated:

DB2 Represents either the DB2 licensed program or a particular DB2 subsystem.

OMEGAMON®
Refers to any of the following products:
- IBM Tivoli® OMEGAMON XE for DB2 Performance Expert on z/OS
- IBM Tivoli OMEGAMON XE for DB2 Performance Monitor on z/OS
- IBM DB2 Performance Expert for Multiplatforms and Workgroups
- IBM DB2 Buffer Pool Analyzer for z/OS

C, C++, and C language
Represent the C or C++ programming language.

CICS® Represents CICS Transaction Server for z/OS.

IMS™ Represents the IMS Database Manager or IMS Transaction Manager.

MVS™ Represents the MVS element of the z/OS operating system, which is equivalent to the Base Control Program (BCP) component of the z/OS operating system.

RACF®
Represents the functions that are provided by the RACF component of the z/OS Security Server.

Accessibility features for DB2 11 for z/OS

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use information technology products successfully.

Accessibility features

The following list includes the major accessibility features in z/OS products, including DB2 11 for z/OS. These features support:
- Keyboard-only operation.
- Interfaces that are commonly used by screen readers and screen magnifiers.
- Customization of display attributes such as color, contrast, and font size

Tip: The Information Management Software for z/OS Solutions Information Center (which includes information for DB2 11 for z/OS) and its related publications are accessibility-enabled for the IBM Home Page Reader. You can operate all features using the keyboard instead of the mouse.

Keyboard navigation

You can access DB2 11 for z/OS ISPF panel functions by using a keyboard or keyboard shortcut keys.

For information about navigating the DB2 11 for z/OS ISPF panels using TSO/E or ISPF, refer to the *z/OS TSO/E Primer*, the *z/OS TSO/E User's Guide*, and the *z/OS ISPF User's Guide*. These guides describe how to navigate each interface, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

Related accessibility information

Online documentation for DB2 11 for z/OS is available in the Information Management Software for z/OS Solutions Information Center, which is available at the following website: http://pic.dhe.ibm.com/infocenter/dzichelp/v2r2/index.jsp

IBM and accessibility

See the *IBM Accessibility Center* at http://www.ibm.com/able for more information about the commitment that IBM has to accessibility.

How to send your comments

Your feedback helps IBM to provide quality information. Please send any comments that you have about this book or other DB2 for z/OS documentation. You can use the following methods to provide comments:

- Send your comments by email to db2zinfo@us.ibm.com and include the name of the product, the version number of the product, and the number of the book. If you are commenting on specific text, please list the location of the text (for example, a chapter and section title or a help topic title).
- You can also send comments by using the **Feedback** link at the footer of each page in the Information Management Software for z/OS Solutions Information Center at http://pic.dhe.ibm.com/infocenter/dzichelp/v2r2/index.jsp.

How to read syntax diagrams

Certain conventions apply to the syntax diagrams that are used in IBM documentation.

Apply the following rules when reading the syntax diagrams that are used in DB2 for z/OS documentation:

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  The ▶▶— symbol indicates the beginning of a statement.

  The —▶ symbol indicates that the statement syntax is continued on the next line.

  The ▶— symbol indicates that a statement is continued from the previous line.

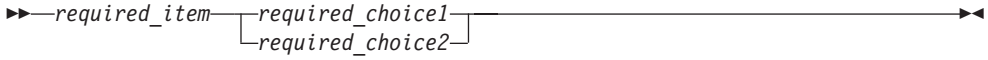
  The —▶◀ symbol indicates the end of a statement.
- Required items appear on the horizontal line (the main path).

- Optional items appear below the main path.

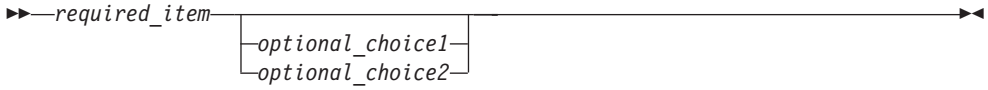
  If an optional item appears above the main path, that item has no effect on the execution of the statement and is used only for readability.

- If you can choose from two or more items, they appear vertically, in a stack.

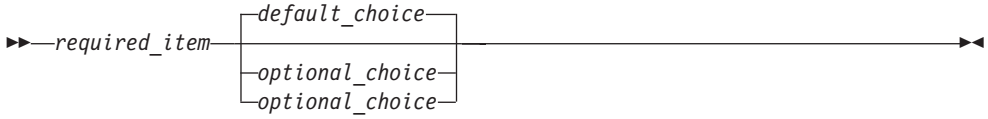
  If you *must* choose one of the items, one item of the stack appears on the main path.



  If choosing one of the items is optional, the entire stack appears below the main path.

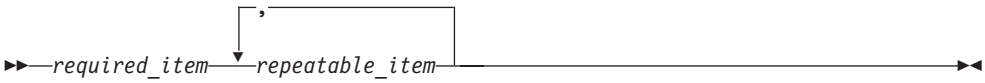


  If one of the items is the default, it appears above the main path and the remaining choices are shown below.



- An arrow returning to the left, above the main line, indicates an item that can be repeated.

If the repeat arrow contains a comma, you must separate repeated items with a comma.



A repeat arrow above a stack indicates that you can repeat the items in the stack.

- Sometimes a diagram must be split into fragments. The syntax fragment is shown separately from the main syntax diagram, but the contents of the fragment should be read as if they are on the main path of the diagram.

fragment-name:

- With the exception of XPath keywords, keywords appear in uppercase (for example, FROM). Keywords must be spelled exactly as shown. XPath keywords are defined as lowercase names, and must be spelled exactly as shown. Variables appear in all lowercase letters (for example, *column-name*). They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

Conventions for describing mixed data values

When mixed data values are shown in examples, certain conventions are used to represent these values.

At sites using a double-byte character set (DBCS), character strings can include a mixture of single-byte and double-byte characters. When mixed data values are shown in the examples, the conventions shown in the following example apply:

Convention	Representation
$^{S}_{O}$	“shift-out” control character (X'0E'), used only for EBCDIC data
$^{S}_{I}$	“shift-in” control character (X'0F'), used only for EBCDIC data
sbcs-string	SBCS string of zero or more single-byte characters
dbcs-string	DBCS string of zero or more double-byte characters
	DBCS apostrophe
	DBCS uppercase G

Figure 1. Conventions used when mixed data values are shown in examples

Industry standards

DB2 for z/OS is developed based on specific industry standards for SQL.

- *ISO/IEC FCD 9075-1:2003, Information technology - Database languages - SQL - Part 1: Framework (SQL/Framework)*
- *ISO/IEC FCD 9075-2:2003, Information technology - Database languages - SQL - Part 2: Foundation (SQL/Foundation)*
- *ISO/IEC FCD 9075-3:2003, Information technology - Database languages - SQL - Part 3: Call-Level Interface (SQL/CLI)*
- *ISO/IEC FCD 9075-4:2003, Information technology - Database languages - SQL - Part 4: Persistent Stored Modules (SQL/PSM)*
- *ISO/IEC FCD 9075-5:2003, Information technology - Database languages - SQL- Part 5: Host Language Bindings (SQL/Bindings)*
- *ISO/IEC FCD 9075-9:2003, Information technology - Database languages - SQL - Part 9: Management of External Data (SQL/MED)*
- *ISO/IEC FCD 9075-10:2003, Information technology - Database languages - SQL - Part 10: Object Language Bindings (SQL/OLB)*
- *ISO/IEC FCD 9075-11:2003, Information technology - Database languages - SQL - Part 11: Information and Definition Schemas (SQL/Schemata)*
- *ISO/IEC FCD 9075-13:2003, Information technology - Database languages - SQL - Part 13: Java Routines and Types (SQL/JRT)*
- *ISO/IEC FCD 9075-14:2006, Information technology - Database languages - SQL - Part 14: XML-Related Specifications (SQL/XML)*
- *ANSI (American National Standards Institute) X3.135-1999, Database Language - SQL*

Chapter 1. DB2 concepts

Certain DB2 concepts are important to understand when using Structured Query Language (SQL).

The following topics provide information on these concepts:

Structured query language

The language that you use to access the data in DB2 tables is the *structured query language* (SQL). SQL is a standardized language for defining and manipulating data in a relational database.

The language consists of SQL statements. SQL statements let you accomplish the following actions:

- Define, modify, or drop data objects, such as tables.
- Retrieve, insert, update, or delete data in tables.

Other SQL statements let you authorize users to access specific resources, such as tables or views.

When you write an SQL statement, you specify what you want done, not how to do it. To access data, for example, you need only to name the tables and columns that contain the data. You do not need to describe how to get to the data.

In accordance with the relational model of data:

- The database is perceived as a set of tables.
- Relationships are represented by values in tables.

- Data is retrieved by using SQL to specify a *result table* that can be derived from one or more tables.

DB2 transforms each SQL statement, that is, the specification of a result table, into a sequence of operations that optimize data retrieval. This transformation occurs when the SQL statement is *prepared*. This transformation is also known as *binding*.

All executable SQL statements must be prepared before they can run. The result of preparation is the executable or *operational form* of the statement.

As the following example illustrates, SQL is generally intuitive. GUPI

Example: Assume that you are shopping for shoes and you want to know what shoe styles are available in size 8. The SQL query that you need to write is similar to the question that you would ask a salesperson, "What shoe styles are available in size 8?" Just as the salesperson checks the shoe inventory and returns with an answer, DB2 retrieves information from a table (SHOES) and returns a result table. The query looks like this:

```
SELECT STYLE
  FROM SHOES
  WHERE SIZE = 8;
```

Assume that the answer to your question is that two shoe styles are available in a size 8: loafers and sandals. The result table looks like this:

```
STYLE
=======
LOAFERS
SANDALS
```

GUPI

You can send an SQL statement to DB2 in several ways. One way is interactively, by entering SQL statements at a keyboard. Another way is through an application program. The program can contain SQL statements that are statically embedded in the application. Alternatively the program can create its SQL statements dynamically, for example, in response to information that a user provides by filling in a form. In this information, you can read about each of these methods.

- Application programming for DB2 (Introduction to DB2 for z/OS)
- SQL: The language of DB2 (Introduction to DB2 for z/OS)
- Programming applications for performance (DB2 Performance)

Static SQL

The source form of a *static* SQL statement is embedded within an application program written in a host language such as COBOL. The statement is prepared before the program is executed and the operational form of the statement persists beyond the execution of the program.

Static SQL statements in a source program must be processed before the program is compiled. This processing can be accomplished through the DB2 precompiler or the DB2 coprocessor. The DB2 precompiler or the coprocessor checks the syntax of the SQL statements, turns them into host language comments, and generates host language statements to invoke DB2.

The preparation of an SQL application program includes precompilation, the preparation of its static SQL statements, and compilation of the modified source program.

Dynamic SQL

Programs that contain embedded *dynamic* SQL statements must be precompiled like those that contain static SQL, but unlike static SQL, the dynamic statements are constructed and prepared at run time.

The source form of a dynamic statement is a character string that is passed to DB2 by the program using the static SQL PREPARE or EXECUTE IMMEDIATE statement. A statement that is prepared using the PREPARE statement can be referenced in a DECLARE CURSOR, DESCRIBE, or EXECUTE statement. Whether the operational form of the statement is persistent depends on whether dynamic statement caching is enabled.

SQL statements embedded in a REXX application are dynamic SQL statements. SQL statements submitted to an interactive SQL facility and to the CALL Level Interface (CLI) are also dynamic SQL.

Deferred embedded SQL

A *deferred embedded* SQL statement is neither fully static nor fully dynamic.

Like a static statement, it is embedded within an application, but like a dynamic statement, it is prepared during the execution of the application. Although prepared at run time, a deferred embedded SQL statement is processed with bind-time rules such that the authorization ID and qualifier determined at bind time for the plan or package owner are used.

Interactive SQL

Interactive SQL refers to SQL statements submitted using SPUFI (SQL processor using file input) or the command line processor.

SPUFI and the command line processor prepares and executes these statements dynamically.

Related concepts:

Command line processor (DB2 Commands)

Related tasks:

Executing SQL by using SPUFI (DB2 Application programming and SQL)

SQL Call Level Interface and Open Database Connectivity

The DB2 Call Level Interface (CLI) is an application programming interface in which functions are provided to application programs to process dynamic SQL statements.

DB2 CLI allows users to access SQL functions directly through a call interface. CLI programs can also be compiled using an Open Database Connectivity (ODBC) Software Developer's Kit, available from Microsoft or other vendors, enabling access to ODBC data sources. Unlike using embedded SQL, no precompilation is required. Applications developed using this interface can be executed on a variety of databases without being compiled against each of databases. Through the

interface, applications use procedure calls at execution time to connect to databases, to issue SQL statements, and to get returned data and status information.

Java database connectivity and embedded SQL for Java

DB2 provides two standards-based Java™ programming APIs: Java Database Connectivity (JDBC) and embedded SQL for Java (SQL/OLB or SQLJ). Both can be used to create Java applications and applets that access DB2.

Static SQL cannot be used by JDBC. SQLJ applications use JDBC as a foundation for such tasks as connecting to databases and handling SQL errors, but can contain embedded static SQL statements in the SQLJ source files. An SQLJ file has to be translated with the SQLJ translator before the resulting Java source code can be compiled.

DB2 data structures

Data structures are elements that are required to use DB2. You can access and use these elements to organize your data. Examples of data structures include tables, table spaces, indexes, index spaces, keys, views, and databases.

The brief descriptions here show how the structures fit into an overall view of DB2. The following figure shows how some DB2 structures contain others. To some extent, the notion of “containment” provides a hierarchy of structures.

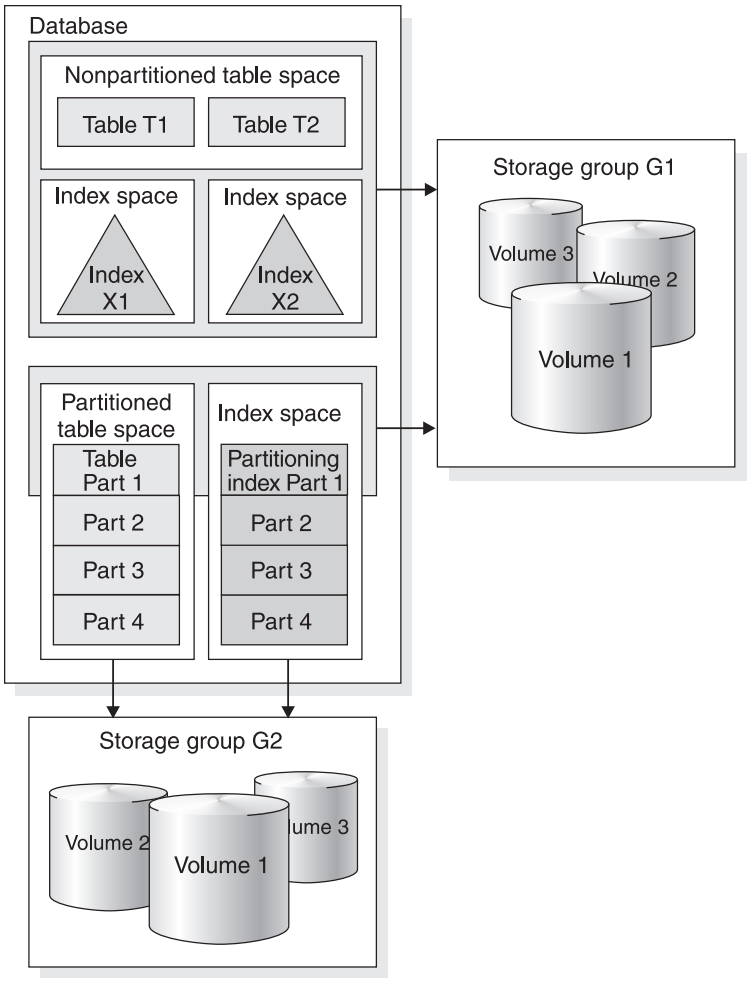


Figure 2. A hierarchy of DB2 structures

The DB2 structures from the most to the least inclusive are:

Databases
A set of DB2 structures that include a collection of tables, their associated indexes, and the table spaces in which they reside.

Storage groups
A set of volumes on disks that hold the data sets in which tables and indexes are stored.

Table spaces
A set of volumes on disks that hold the data sets in which tables and indexes are stored.

Tables All data in a DB2 database is presented in *tables*, which are collections of rows all having the same columns. A table that holds persistent user data is a *base table*. A table that stores data temporarily is a *temporary table*.

Views A *view* is an alternative way of representing data that exists in one or more tables. A view can include all or some of the columns from one or more base tables.

Indexes
An *index* is an ordered set of pointers to the data in a DB2 table. The index is stored separately from the table.

Related concepts:

"DB2 system objects" on page 18

Implementing your database design (DB2 Administration Guide)

"Storage structures" on page 16

DB2 tables

Tables are logical structures that DB2 maintains. DB2 supports several different types of tables.

Tables are made up of columns and rows. The rows of a relational table have no fixed order. The order of the columns, however, is always the order in which you specified them when you defined the table.

At the intersection of every column and row is a specific data item, which is called a *value*. A *column* is a set of values of the same type. A *row* is a sequence of values such that the *n*th value is a value of the *n*th column of the table. Every table must have one or more columns, but the number of rows can be zero.

DB2 accesses data by referring to its content instead of to its location or organization in storage.

DB2 supports several different types of tables:

- Archive tables
- Archive-enabled tables
- Auxiliary tables
- Base tables
- Clone tables
- Empty tables
- History tables
- Materialized query tables
- Result tables
- Temporal tables
- Temporary tables
- XML tables

Creation of tables (Introduction to DB2 for z/OS)

Types of tables (Introduction to DB2 for z/OS)

DB2 indexes

An *index* is an ordered set of pointers to rows of a table. DB2 can use indexes to improve performance and ensure uniqueness. Understanding the structure of DB2 indexes can help you achieve the best performance for your system.

Conceptually, you can think of an index to the rows of a DB2 table like you think of an index to the pages of a book. Each index is based on the values of data in one or more columns of a table.

DB2 can use indexes to ensure uniqueness and to improve performance by clustering data, partitioning data, and providing efficient access paths to data for queries. In most cases, access to data is faster with an index than with a scan of the data. For example, you can create an index on the DEPTNO column of the sample DEPT table to easily locate a specific department and avoid reading through each row of, or *scanning*, the table.

An index is stored separately from the data in the table. Each index is physically stored in its own index space. When you define an index by using the CREATE INDEX statement, DB2 builds this structure and maintains it automatically. However, you can perform necessary maintenance such as reorganizing it or recovering the index.

The main purposes of indexes are:

- To improve performance. Access to data is often faster with an index than without.
- To ensure that a row is unique. For example, a unique index on the employee table ensures that no two employees have the same employee number.
- To cluster the data.
- To determine which partition the data goes into.
- To provide index-only access to data.

Except for changes in performance, users of the table are unaware that an index is in use. DB2 decides whether to use the index to access the table. Some techniques enable you to influence how indexes affect performance when you calculate the storage size of an index and determine what type of index to use.

An index can be either partitioning or nonpartitioning, and either type can be clustered. For example, you can apportion data by last names, possibly using one partition for each letter of the alphabet. Your choice of a partitioning scheme is based on how an application accesses data, how much data you have, and how large you expect the total amount of data to grow.

Be aware that indexes have both benefits and disadvantages. A greater number of indexes can simultaneously improve the access performance of a particular transaction and require additional processing for inserting, updating, and deleting index keys. After you create an index, DB2 maintains the index, but you can perform necessary maintenance, such as reorganizing it or recovering it, as necessary.

Creation of indexes (Introduction to DB2 for z/OS)
"CREATE INDEX" on page 1267

DB2 keys

A *key* is a column or an ordered collection of columns that is identified in the description of a table, an index, or a referential constraint. Keys are crucial to the table structure in a relational database.

Keys are important in a relational database because they ensure that each record in a table is uniquely identified, they help establish and enforce referential integrity, and they establish relationships between tables. The same column can be part of more than one key.

A *composite key* is an ordered set of two or more columns of the same table. The ordering of the columns is not constrained by their actual order within the table. The term *value*, when used with respect to a composite key, denotes a composite value. For example, consider this rule: "The value of the foreign key must be equal to the value of the primary key." This rule means that each component of the value of the foreign key must be equal to the corresponding component of the value of the primary key.

DB2 supports several types of keys.

Unique keys

A *unique constraint* is a rule that the values of a key are valid only if they are unique. A key that is constrained to have unique values is a *unique key*. DB2 uses a *unique index* to enforce the constraint during the execution of the LOAD utility and whenever you use an INSERT, UPDATE, or MERGE statement to add or modify data. Every unique key is a key of a unique index. You can define a unique key by using the UNIQUE clause of either the CREATE TABLE or the ALTER TABLE statement. A table can have any number of unique keys.

The columns of a unique key cannot contain null values.

Primary keys

A *primary key* is a special type of unique key and cannot contain null values. For example, the DEPTNO column in the DEPT table is a primary key.

A table can have no more than one primary key. Primary keys are optional and can be defined in CREATE TABLE or ALTER TABLE statements.

The unique index on a primary key is called a *primary index*. When a primary key is defined in a CREATE TABLE statement or ALTER TABLE statement, DB2 automatically creates the primary index if one of the following conditions is true:

- DB2 is operating in new-function mode, and the table space is implicitly created.
- DB2 is operating in new-function mode, the table space is explicitly created, and the schema processor is running.
- DB2 is operating in conversion mode, and the schema processor is running.

If a unique index already exists on the columns of the primary key when it is defined in the ALTER TABLE statement, this unique index is designated as the primary index when DB2 is operating in new-function mode and implicitly created the table space.

Parent keys

A *parent key* is either a primary key or a unique key in the parent table of a referential constraint. The values of a parent key determine the valid values of the foreign key in the constraint.

Foreign keys

A *foreign key* is a key that is specified in the definition of a referential constraint in a CREATE or ALTER TABLE statement. A foreign key refers to or is related to a specific parent key.

Unlike other types of keys, a foreign key does not require an index on its underlying column or columns. A table can have zero or more foreign keys. The value of a composite foreign key is null if any component of the value is null.

The following figure shows the relationship between some columns in the DEPT table and the EMP table.

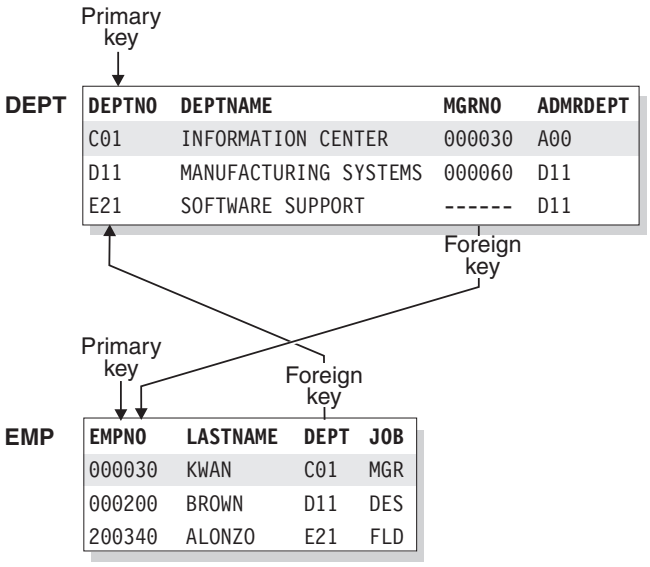


Figure 3. Relationship between DEPT and EMP tables

Figure notes: Each table has a primary key:
- DEPTNO in the DEPT table
- EMPNO in the EMP table

Each table has a foreign key that establishes a relationship between the tables:
- The values of the foreign key on the DEPT column of the EMP table match values in the DEPTNO column of the DEPT table.
- The values of the foreign key on the MGRNO column of the DEPT table match values in the EMPNO column of the EMP table when an employee is a manager.

To see a specific relationship between rows, notice how the shaded rows for department C01 and employee number 000030 share common values.

Related concepts:

"Referential constraints" on page 23

DB2 views

A *view* is an alternative way of representing data that exists in one or more tables. A view can include all or some of the columns from one or more base tables.

A view is a named specification of a result table. Conceptually, creating a view is somewhat like using binoculars. You might look through binoculars to see an entire landscape or to look at a specific image within the landscape, such as a tree.

You can create a view that:
- Combines data from different base tables
- Is based on other views or on a combination of views and tables
- Omits certain data, thereby shielding some table data from users

In fact, these are common underlying reasons to use a view. Combining information from base tables and views simplifies retrieving data for a user, and limiting the data that a user can see is useful for security. You can use views for a number of different purposes. A view can:
- Control access to a table

- Make data easier to use
- Simplify authorization by granting access to a view without granting access to the table
- Show only portions of data in the table
- Show summary data for a given table
- Combine two or more tables in meaningful ways
- Show only the selected rows that are pertinent to the process that uses the view

To define a view, you use the CREATE VIEW statement and assign a name (up to 128 characters in length) to the view. Specifying the view in other SQL statements is effectively like running an SQL SELECT statement. At any time, the view consists of the rows that would result from the SELECT statement that it contains. You can think of a view as having columns and rows just like the base table on which the view is defined.

You also can specify a period specification for a view, subject to certain restrictions.

Example

GUPI

Example 1: The following figure shows a view of the EMP table that omits sensitive employee information and renames some of the columns.

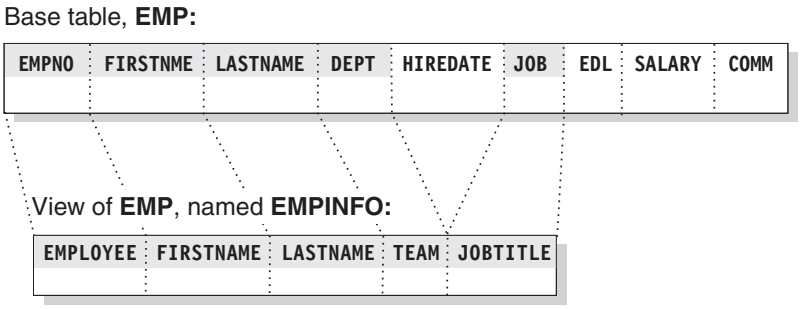


Figure 4. A view of the EMP table

Figure note: The EMPINFO view represents a table that includes columns named EMPLOYEE, FIRSTNAME, LASTNAME, TEAM, and JOBTITLE. The data in the view comes from the columns EMPNO, FIRSTNME, LASTNAME, DEPT, and JOB of the EMP table.

Example 2: The following CREATE VIEW statement defines the EMPINFO view that is shown in the preceding figure:

```
CREATE VIEW EMPINFO (EMPLOYEE, FIRSTNAME, LASTNAME, TEAM, JOBTITLE)
  AS SELECT EMPNO, FIRSTNME, LASTNAME, DEPT, JOB
  FROM EMP;
```

When you define a view, DB2 stores the definition of the view in the DB2 catalog. However, DB2 does not store any data for the view itself, because the data exists in the base table or tables.

Example 3: You can narrow the scope of the EMPINFO view by limiting the content to a subset of rows and columns that includes departments A00 and C01 only:

```
CREATE VIEW EMPINFO (EMPLOYEE, FIRSTNAME, LASTNAME, TEAM, JOBTITLE)
  AS SELECT EMPNO, FIRSTNME, LASTNAME, DEPT, JOB
  WHERE DEPT = 'AOO' OR DEPT = 'C01'
    FROM EMP;
```

GUPI

In general, a view inherits the attributes of the object from which it is derived. Columns that are added to the tables after the view is defined on those tables do not appear in the view.

Restriction: You cannot create an index for a view. In addition, you cannot create any form of a key or a constraint (referential or otherwise) on a view. Such indexes, keys, or constraints must be built on the tables that the view references.

To retrieve or access information from a view, you use views like you use base tables. You can use a SELECT statement to show the information from the view. The SELECT statement can name other views and tables, and it can use the WHERE, GROUP BY, and HAVING clauses. It cannot use the ORDER BY clause or name a host variable.

Whether a view can be used in an insert, update, or delete operation depends on its definition. For example, if a view includes a foreign key of its base table, INSERT and UPDATE operations that use the view are subject to the same referential constraint as the base table. Likewise, if the base table of a view is a parent table, DELETE operations that use the view are subject to the same rules as DELETE operations on the base table. *Read-only* views cannot be used for insert, update, and delete operations.

Related information:

- Implementing DB2 views (DB2 Administration Guide)
  - Creation of views (Introduction to DB2 for z/OS)
  - Referential constraints (Introduction to DB2 for z/OS)
  - Employee table (DSN8B10.EMP) (Introduction to DB2 for z/OS)

"CREATE VIEW" on page 1527

DB2 schemas and schema qualifiers

The objects in a relational database are organized into sets called schemas. A *schema* is a collection of named objects that provides a logical classification of objects in the database. The first part of a schema name is the qualifier.

A schema provides a logical classification of objects in the database. The objects that a schema can contain include tables, indexes, table spaces, distinct types, functions, stored procedures, and triggers. An object is assigned to a schema when it is created.

The *schema name* of the object determines the schema to which the object belongs. A user object, such as a distinct type, function, procedure, sequence, or trigger should not be created in a *system schema*, which is any one of a set of schemas that are reserved for use by the DB2 subsystem.

When a table, index, table space, distinct type, function, stored procedure, or trigger is created, it is given a qualified two-part name. The first part is the schema

name (or the qualifier), which is either implicitly or explicitly specified. The default schema is the authorization ID of the owner of the plan or package. The second part is the name of the object.

In previous versions, CREATE statements had certain restrictions when the value of CURRENT SCHEMA was different from CURRENT SQLID value. Although those restrictions no longer exist, you now must consider how to determine the qualifier and owner when CURRENT SCHEMA and CURRENT SQLID contain different values. The rules for how the owner is determined depend on the type of object being created.

CURRENT SCHEMA and CURRENT SQLID affect only dynamic SQL statements. Static CREATE statements are not affected by either CURRENT SCHEMA or CURRENT SQLID.

The following table summarizes the effect of CURRENT SCHEMA in determining the schema qualifier and owner for these objects:

- Alias
- Auxiliary table
- Created global temporary table
- Table
- View

Table 1. Schema qualifier and owner for objects

Specification of name for new object being created	Schema qualifier of new object	Owner of new object
name (no qualifier)	value of CURRENT SCHEMA	value of CURRENT SQLID
abc.name (single qualifier)	abc	abc
.....abc.name (multiple qualifiers)	abc	abc

The following table summarizes the effect of CURRENT SCHEMA in determining the schema qualifier and owner for these objects:

- User-defined distinct type
- User-defined function
- Procedure
- Sequence
- Trigger

Table 2. Schema qualifier and owner for additional objects

Specification of name for new object being created	Schema qualifier of new object	Owner of new object
name (no qualifier)	value of CURRENT SCHEMA	value of CURRENT SQLID
abc.name (single qualifier)	abc	value of CURRENT SQLID
.....abc.name (multiple qualifiers)	abc	value of CURRENT SQLID

"Reserved schema names" on page 2020

DB2 storage groups

DB2 *storage groups* are a set of volumes on disks that hold the data sets in which tables and indexes are stored.

The description of a storage group names the group and identifies its volumes and the VSAM (Virtual Storage Access Method) catalog that records the data sets. The default storage group, SYSDEFLT, is created when you install DB2.

Within the storage group, DB2 does the following actions:

- Allocates storage for table spaces and indexes
- Defines the necessary VSAM data sets
- Extends and deletes VSAM data sets
- Alters VSAM data sets

All volumes of a given storage group must have the same device type. However, parts of a single database can be stored in different storage groups.

DB2 can manage the auxiliary storage requirements of a database by using DB2 storage groups. Data sets in these DB2 storage groups are called *DB2-managed data sets*.

These DB2 storage groups are not the same as storage groups that are defined by the DFSMS storage management subsystem (DFSMSsms).

You have several options for managing DB2 data sets:

- Let DB2 manage the data sets. This option means less work for DB2 database administrators.

  After you define a DB2 storage group, DB2 stores information about it in the DB2 catalog. (This catalog is not the same as the integrated catalog facility catalog that describes DB2 VSAM data sets). The catalog table SYSIBM.SYSSTOGROUP has a row for each storage group, and SYSIBM.SYSVOLUMES has a row for each volume. With the proper authorization, you can retrieve the catalog information about DB2 storage groups by using SQL statements.

  When you create table spaces and indexes, you name the storage group from which space is to be allocated. You can also assign an entire database to a storage group. Try to assign frequently accessed objects (indexes, for example) to fast devices, and assign seldom-used tables to slower devices. This approach to choosing storage groups improves performance.

  If you are authorized and do not take specific steps to manage your own storage, you can still define tables, indexes, table spaces, and databases. A default storage group, SYSDEFLT, is defined when DB2 is installed. DB2 uses SYSDEFLT to allocate the necessary auxiliary storage. Information about SYSDEFLT, as with any other storage group, is kept in the catalog tables SYSIBM.SYSSTOGROUP and SYSIBM.SYSVOLUMES.

  For both user-managed and DB2-managed data sets, you need at least one integrated catalog facility (ICF) catalog; this catalog can be either a user catalog or a master catalog. These catalogs are created with the ICF. You must identify the catalog of the ICF when you create a storage group or when you create a table space that does not use storage groups.
- Let SMS manage some or all the data sets, either when you use DB2 storage groups or when you use data sets that you have defined yourself. This option

offers a reduced workload for DB2 database administrators and storage administrators. You can specify SMS classes when you create or alter a storage group.
- Define and manage your own data sets using VSAM Access Method Services. This option gives you the most control over the physical storage of tables and indexes.

Recommendation: Use DB2 storage groups and whenever you can, either specifically or by default. Also use SMS managed DB2 storage groups whenever you can.

Related tasks:

Choosing data page sizes for LOB data (DB2 Performance)

DB2 databases

DB2 *databases* are a set of DB2 structures that include a collection of tables, their associated indexes, and the table spaces in which they reside. You define a database by using the CREATE DATABASE statement.

Whenever a table space is created, it is explicitly or implicitly assigned to an existing database. If you create a table space and do not specify a database name, the table space is created in the default database, DSNDB04. In this case, DB2 implicitly creates a database or uses an existing implicitly created database for the table. All users who have the authority to create table spaces or tables in database DSNDB04 have authority to create tables and table spaces in an implicitly created database. If the table space is implicitly created, and you do not specify the IN clause in the CREATE TABLE statement, DB2 implicitly creates the database to which the table space is assigned.

A single database, for example, can contain all the data that is associated with one application or with a group of related applications. Collecting that data into one database allows you to start or stop access to all the data in one operation. You can also grant authorization for access to all the data as a single unit. Assuming that you are authorized to access data, you can access data that is stored in different databases.

Recommendation: Keep only a minimal number of table spaces in each database, and a minimal number of tables in each table space. Excessive numbers of table spaces and tables in a database can cause decreases in performance and manageability issues. If you reduce the number of table spaces and tables in a database, you improve performance, minimize maintenance, increase concurrency, and decrease log volume.

The following figure shows how the main DB2 data structures fit together. Two databases, A and B, are represented as squares. Database A contains a table space and two index spaces. The table space is segmented and contains tables A1 and A2. Each index space contains one index, an index on table A1 and an index on table A2. Database B contains one table space and one index space. The table space is partitioned and contains table B1, partitions 1 through 4. The index space contains one partitioning index, parts 1 - 4.

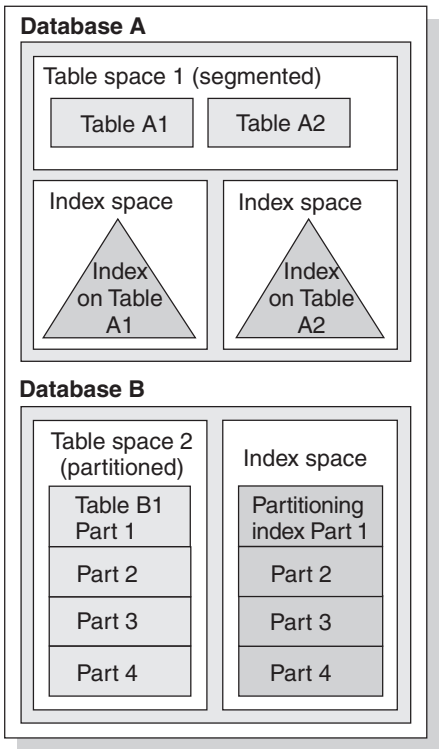


Figure 5. Data structures in a DB2 database

When you migrate to the current version, DB2 adopts the default database and default storage group that you used in the previous version. You have the same authority for the current version as you did in the previous version.

Reasons to define a database

In DB2 for z/OS, a database is a logical collection of table spaces and index spaces. Consider the following factors when deciding whether to define a new database for a new set of objects:

- You can start and stop an entire database as a unit; you can display the statuses of all its objects by using a single command that names only the database. Therefore, place a set of tables that are used together into the same database. (The same database holds all indexes on those tables.)
- Some operations lock an entire database. For example, some phases of the LOAD utility prevent some SQL statements (CREATE, ALTER, and DROP) from using the same database concurrently. Therefore, placing many unrelated tables in a single database is often inconvenient.

  When one user is executing a CREATE, ALTER, or DROP statement for a table, no other user can access the database that contains that table. QMF™ users, especially, might do a great deal of data definition; the QMF operations SAVE DATA and ERASE *data-object* are accomplished by creating and dropping DB2 tables. For maximum concurrency, create a separate database for each QMF user.
- The internal database descriptors (DBDs) might become inconveniently large. DBDs grow as new objects are defined, but they do not immediately shrink when objects are dropped; the DBD space for a dropped object is not reclaimed until the MODIFY RECOVERY utility is used to delete records of obsolete copies

from SYSIBM.SYSCOPY. DBDs occupy storage and are the objects of occasional input and output operations. Therefore, limiting the size of DBDs is another reason to define new databases.

Creation of databases (Introduction to DB2 for z/OS)

Storage structures

In DB2, a *storage structure* is a set of one or more VSAM data sets that hold DB2 tables or indexes. A storage structure is also called a *page set*.

The two primary types of storage structures in DB2 for z/OS are table spaces and index spaces.

Related concepts:

"DB2 data structures" on page 4

Related information:

Implementing DB2 table spaces (DB2 Administration Guide)

Implementing DB2 indexes (DB2 Administration Guide)

DB2 table spaces

A DB2 *table space* is a set of volumes on disks that hold the data sets in which tables are actually stored. All tables are kept in table spaces. A table space can have one or more tables.

A table space can consist of a number of VSAM data sets. Data sets are VSAM linear data sets (LDSs). Table spaces are divided into equal-sized units, called *pages*, which are written to or read from disk in one operation. You can specify page sizes (4 KB, 8 KB, 16 KB, or 32 KB in size) for the data; the default page size is 4 KB. As a general rule, you should have only one table in each table space. It is also best to keep only one table space in each database. If you must have more than one table space in a database, keep no more than 20 table spaces in that database.

Data in most table spaces can be compressed, which can allow you to store more data on each data page.

You can explicitly define a table space by using the CREATE TABLESPACE statement, which can specify the database to which the table space belongs and the storage group that it uses.

Alternatively, you can let DB2 implicitly create a table space for you by issuing a CREATE TABLE statement that does not specify an existing table space. In this case, DB2 assigns the table space to the default database and the default storage group. If DB2 is operating in conversion mode, a segmented table space is created. In new-function mode, DB2 creates a partition-by-growth table space.

The maximum number of partitions for a table space depends on the page size and on the DSSIZE. The size of the table space depends on how many partitions are in the table space and on the DSSIZE. The maximum number of partitions for a partition-by-growth table space depends on the value that is specified for the MAXPARTITIONS option of the CREATE TABLESPACE or ALTER TABLESPACE statement.

When you create a table space, you can specify what type of table space is created. DB2 supports different types of table spaces:

Universal table spaces
Provide better space management (for varying-length rows) and improved mass delete performance by combining characteristics of partitioned and segmented table space schemes. A universal table space can hold one table.

Partitioned table spaces
Divide the available space into separate units of storage called *partitions*. Each partition contains one data set of one table.

Segmented table spaces
Divide the available space into groups of pages called *segments*. Each segment is the same size. A segment contains rows from only one table.

Large object table spaces
Hold large object data such as graphics, video, or very large text strings. A LOB table space is always associated with the table space that contains the logical LOB column values.

Simple table spaces
Can contain more than one table. The rows of different tables are not kept separate (unlike segmented table spaces).

Restriction: Starting in DB2 Version 9.1, you cannot create a simple table space. Simple table spaces that were created with an earlier version of DB2 are still supported.

XML table spaces
Hold XML data. An XML table space is always associated with the table space that contains the logical XML column value.

Related tasks:

Choosing data page sizes (DB2 Performance)

Related reference:

Examples of table space definitions (DB2 Administration Guide)

"ALTER TABLESPACE" on page 1074

"CREATE TABLESPACE" on page 1455

Related information:

Implementing DB2 table spaces (DB2 Administration Guide)

DB2 index spaces

An *index space* is a DB2 storage structure that contains a single index.

When you create an index by using the CREATE INDEX statement, an index space is automatically defined in the same database as the table. You can define a unique name for the index space, or DB2 can derive a unique name for you. Under certain circumstances, DB2 implicitly creates index spaces.

DB2 hash spaces

A *hash space* is a defined disk space that organizes table data for hash access.

When you enable hash access on a table, DB2 requires a defined amount of disk space to contain table data. You can specify the amount of disk space to allocate to the hash space when you create a table or alter an existing table. The hash space on a table must be large enough to contain new rows that are added to the table. If a hash space is full, new rows are relocated to the overflow index, which reduces

the performance of hash access on that table. Hash spaces can contain only a single table in a universal table space, and can be partitioned by range or partitioned by growth.

Related tasks:

- Managing space and page size for hash-organized tables (DB2 Performance)
- Fine-tuning hash space and page size (DB2 Performance)
- Creating tables that use hash organization (DB2 Administration Guide)
- Altering tables to enable hash access (DB2 Administration Guide)
- Organizing tables by hash for fast access to individual rows (DB2 Performance)
- Monitoring hash access (DB2 Performance)
  - Database design with hash access (Introduction to DB2 for z/OS)
  - Hash access paths (Introduction to DB2 for z/OS)

DB2 system objects

Unlike the DB2 data structures that users create and access, DB2 controls and accesses system objects.

DB2 has a comprehensive infrastructure that enables it to provide data integrity, performance, and the ability to recover user data. In addition, Parallel Sysplex® data sharing uses shared system objects.

Related concepts:

"DB2 data structures" on page 4

DB2 catalog

DB2 maintains a set of tables that contain information about the data that DB2 controls. These tables are collectively known as the *catalog*.

The catalog tables contain information about DB2 objects such as tables, views, and indexes. When you create, alter, or drop an object, DB2 inserts, updates, or deletes rows of the catalog that describe the object.

The DB2 catalog consists of tables of data about everything defined to the DB2 system, including table spaces, indexes, tables, copies of table spaces and indexes, and storage groups. The system database DSNDB06 contains the DB2 catalog.

When you create, alter, or drop any structure, DB2 inserts, updates, or deletes rows of the catalog that describe the structure and tell how the structure relates to other structures. For example, SYSIBM.SYSTABLES is one catalog table that records information when a table is created. DB2 inserts a row into SYSIBM.SYSTABLES that includes the table name, its owner, its creator, and the name of its table space and its database.

To understand the role of the catalog, consider what happens when the EMP table is created. DB2 records the following data:

Table information
To record the table name and the name of its owner, its creator, its type, the name of its table space, and the name of its database, DB2 inserts a row into the catalog.

Column information
To record information about each column of the table, DB2 inserts the name of the table to which the column belongs, its length, its data type, and its sequence number by inserting a row into the catalog for each column of the table.

Authorization information
To record that the owner of the table has authorization to create the table, DB2 inserts a row into the catalog.

Tables in the catalog are like any other database tables with respect to retrieval. If you have authorization, you can use SQL statements to look at data in the catalog tables in the same way that you retrieve data from any other table in the DB2 database. DB2 ensures that the catalog contains accurate object descriptions. If you are authorized to access the specific tables or views on the catalog, you can SELECT from the catalog, but you cannot use INSERT, UPDATE, DELETE, TRUNCATE, or MERGE statements on the catalog.

The *communications database* (CDB) is part of the DB2 catalog. The CDB consists of a set of tables that establish conversations with remote database management systems (DBMSs). The distributed data facility (DDF) uses the CDB to send and receive distributed data requests.

"DB2 catalog tables" on page 2102

DB2 directory

The DB2 directory contains information that DB2 uses during normal operation.

You cannot access the directory by using SQL, although much of the same information is contained in the DB2 catalog, for which you can submit queries. The structures in the directory are not described in the DB2 catalog.

The directory consists of a set of DB2 tables that are stored in table spaces in system database DSNDB01. Each of the table spaces that are listed in the following table is contained in a VSAM linear data set.

Table 3. Directory table spaces

Table space name	Description
SCT02	Contains the internal form of SQL statements that are contained in an application. If you bound a plan with SQL statements in a prior release, DB2 created a structure in SCT02.
SPT01 Skeleton package	Contains the internal form of SQL statements that are contained in a package.
SYSSPUXA	Contains the contents of a package selection.
SYSSPUXB	Contains the contents of a package explain block.
SYSLGRNX Log range	Tracks the opening and closing of table spaces, indexes, or partitions. By tracking this information and associating it with relative byte addresses (RBAs) as contained in the DB2 log, DB2 can reduce recovery time by reducing the amount of log that must be scanned for a particular table space, index, or partition.

Table 3. Directory table spaces (continued)

Table space name	Description
SYSUTILX System utilities	Contains a row for every utility job that is running. The row persists until the utility is finished. If the utility terminates without completing, DB2 uses the information in the row when you restart the utility.
DBD01 Database descriptor (DBD)	Contains internal information, called *database descriptors* (DBDs), about the databases that exist within the DB2 subsystem. Each database has exactly one corresponding DBD that describes the database, table spaces, tables, table check constraints, indexes, and referential relationships. A DBD also contains other information about accessing tables in the database. DB2 creates and updates DBDs whenever their corresponding databases are created or updated.
SYSDBDXA	Contains the contents of a DBD section.

Active and archive logs

DB2 records all data changes and other significant events in a log.

If you keep these logs, DB2 can re-create those changes for you in the event of a failure or roll the changes back to a previous point in time.

DB2 writes each log record to a disk data set called the *active log*. When the active log is full, DB2 copies the contents of the active log to a disk or magnetic tape data set called the *archive log*.

You can choose either single logging or dual logging.

- A single active log contains up to 93 active log data sets.
- With dual logging, the active log has twice the capacity for active log data sets, because two identical copies of the log records are kept.

Each DB2 subsystem manages multiple active logs and archive logs. The following facts are true about each DB2 active log:

- Each log can be duplexed to ensure high availability.
- Each active log data set is a VSAM linear data set (LDS).
- DB2 supports striped active log data sets.

Related tasks:

Managing the log and the bootstrap data set (DB2 Administration Guide)

Related information:

Reading log records (DB2 Administration Guide)

Bootstrap data set

The *bootstrap data set* (*BSDS*) is a VSAM key-sequenced data set (KSDS). This KSDS contains information that is critical to DB2, such as the names of the logs. DB2 uses information in the BSDS for system restarts and for any activity that requires reading the log.

Specifically, the BSDS contains:

- An inventory of all active and archive log data sets that are known to DB2. DB2 uses this information to track the active and archive log data sets. DB2 also uses this information to locate log records to satisfy log read requests during normal DB2 system activity and during restart and recovery processing.
- A wrap-around inventory of all recent DB2 checkpoint activity. DB2 uses this information during restart processing.
- The distributed data facility (DDF) communication record, which contains information that is necessary to use DB2 as a distributed server or requester.
- Information about buffer pools.

Because the BSDS is essential to recovery in the event of subsystem failure, during installation DB2 automatically creates two copies of the BSDS and, if space permits, places them on separate volumes.

The BSDS can be duplexed to ensure availability.

Related tasks:

Managing the log and the bootstrap data set (DB2 Administration Guide)

Buffer pools

Buffer pools are areas of virtual storage that temporarily store pages of table spaces or indexes.

When an application program accesses a row of a table, DB2 places the page that contains that row in a buffer. Access to data in this temporary storage is faster than accessing data on a disk. If the required data is already in a buffer, the application program does not need to wait for it to be retrieved from disk, so the time and cost of retrieving the page is reduced.

Buffer pools require monitoring and tuning. Buffer pool sizes are critical to the performance characteristics of an application or group of applications that access data in those buffer pools.

You can specify default buffer pools for user data and for indexes. A special type of buffer pool that is used only in Parallel Sysplex data sharing is the *group buffer pool*, which resides in the coupling facility. Group buffer pools reside in a special PR/SM™ LPAR logical partition called a *coupling facility*, which enables several DB2 subsystems to share information and control the coherency of data.

Buffer pools reside in the DB2 DBM1 primary address space. This option offers the best performance. The maximum size of a buffer pool is 1 TB.

Related tasks:

Tuning database buffer pools (DB2 Performance)

Calculating buffer pool size (DB2 Installation and Migration)

Enabling automatic buffer pool size management (DB2 Performance)

Data definition control support database

The *data definition control support* (DDCS) database refers to a user-maintained collection of tables that are used by data definition control support to restrict the submission of specific DB2 DDL (data definition language) statements to selected application identifiers (plans or collections of packages).

This database is automatically created during installation. After this database is created, you must populate the tables to use this facility. The system name for this database is DSNRGFDB.

Resource limit facility tables

The *resource limit facility* enables you to control the amount of processor resources that are used by SQL statements.

GUPI For example, you might choose to disable bind operations during critical times of day to avoid contention with the DB2 catalog.

Resource limits apply to the following types of SQL statements:

- SELECT
- INSERT
- UPDATE
- MERGE
- TRUNCATE
- DELETE

Resource limits apply only to dynamic SQL statements. The resource limit facility does not control static SQL statements regardless of whether they are issued locally or remotely, and no limits apply to primary or secondary authorization IDs that have installation SYSADM or installation SYSOPR authority.

You can establish a single limit for all users, different limits for individual users, or both. You can choose to have these limits applied before the statement is executed through *predictive governing*, or while a statement is running , through *reactive governing*. You can also use reactive and predictive governing in combination. You define these limits in one or more *resource limit tables*, named DSNRLST*xx* or DSNRLMT*xx*, depending on the monitoring purpose. GUPI

Related concepts:

Controlling the resource limit facility (DB2 Administration Guide)

Related tasks:

Setting limits for system resource usage by using the resource limit facility (DB2 Performance)

Limiting resources for SQL statements reactively (DB2 Performance)

Limiting resources for SQL statements predictively (DB2 Performance)

Combining reactive and predictive governing (DB2 Performance)

Related reference:

Resource limit facility tables (DB2 Performance)

Work file database

Use the *work file database* as storage for processing SQL statements that require working space, such as that required for a sort.

The work file database is used as storage for DB2 work files for processing SQL statements that require working space (such as the space that is required for a sort), and as storage for created global temporary tables and declared global temporary tables.

DB2 creates a work file database and some table spaces in it for you at installation time. You can create additional work file table spaces at any time. You can drop, re-create, and alter the work file database or the table spaces in it, or both, at any time.

In a non-data-sharing environment, the work file database is named DSNDB07. In a data sharing environment, each DB2 member in the data sharing group has its own work file database.

You can also use the work file database for all temporary tables.

DB2 and data integrity

Referential integrity ensures data integrity by enforcing rules with referential constraints, check constraints, and triggers. You can rely on constraints and triggers to ensure the integrity and validity of your data, rather than relying on individual applications to do that work.

Constraints

Constraints are rules that control values in columns to prevent duplicate values or set restrictions on data added to a table.

Constraints fall into the following three types:

- Unique constraints
- Referential constraints
- Check constraints

Unique constraints

A *unique constraint* is a rule that the values of a key are valid only if they are unique in a table.

Unique constraints are optional and can be defined in the CREATE TABLE or ALTER TABLE statements with the PRIMARY KEY clause or the UNIQUE clause. The columns specified in a unique constraint must be defined as NOT NULL. A unique index enforces the uniqueness of the key during changes to the columns of the unique constraint.

A table can have an arbitrary number of unique constraints, with at most one unique constraint defined as a primary key. A table cannot have more than one unique constraint on the same set of columns.

A unique constraint that is referenced by the foreign key of a referential constraint is called the *parent key*.

Referential constraints

DB2 ensures referential integrity between your tables when you define referential constraints.

Referential integrity is the state in which all values of all foreign keys are valid. Referential integrity is based on *entity integrity*. Entity integrity requires that each

entity have a unique key. For example, if every row in a table represents relationships for a unique entity, the table should have one column or a set of columns that provides a unique identifier for the rows of the table. This column (or set of columns) is called the parent key of the table. To ensure that the parent key does not contain duplicate values, a unique index must be defined on the column or columns that constitute the parent key. Defining the parent key is called entity integrity.

A *referential constraint* is the rule that the nonnull values of a foreign key are valid only if they also appear as values of a parent key. The table that contains the parent key is called the *parent table* of the referential constraint, and the table that contains the foreign key is a *dependent* of that table.

The relationship between some rows of the DEPT and EMP tables, shown in the following figure, illustrates referential integrity concepts and terminology. For example, referential integrity ensures that every foreign key value in the DEPT column of the EMP table matches a primary key value in the DEPTNO column of the DEPT table.

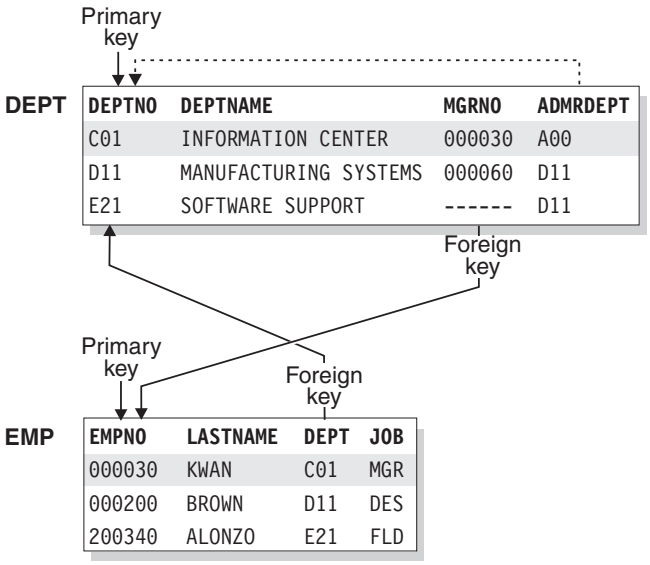


Figure 6. Referential integrity of DEPT and EMP tables

Two parent and dependent relationships exist between the DEPT and EMP tables.

- The foreign key on the DEPT column establishes a parent and dependent relationship. The DEPT column in the EMP table depends on the DEPTNO in the DEPT table. Through this foreign key relationship, the DEPT table is the parent of the EMP table. You can assign an employee to no department (by specifying a null value), but you cannot assign an employee to a department that does not exist.
- The foreign key on the MGRNO column also establishes a parent and dependent relationship. Because MGRNO depends on EMPNO, EMP is the parent table of the relationship, and DEPT is the dependent table.

You can define a primary key on one or more columns. A primary key that includes two or more columns is called a *composite key*. A foreign key can also include one or more columns. When a foreign key contains multiple columns, the corresponding primary key must be a composite key. The number of foreign key columns must be the same as the number of columns in the parent key, and the

data types of the corresponding columns must be compatible. (The sample project activity table, DSN8B10.PROJACT, is an example of a table with a primary key on multiple columns, PROJNO, ACTNO, and ACSTDATE.)

A table can be a dependent of itself; this is called a *self-referencing table*. For example, the DEPT table is self-referencing because the value of the administrative department (ADMRDEPT) must be a department ID (DEPTNO). To enforce the self-referencing constraint, DB2 requires that a foreign key be defined.

Similar terminology applies to the rows of a parent-and-child relationship. A row in a dependent table, called a *dependent row*, refers to a row in a parent table, called a *parent row*. But a row of a parent table is not always a parent row—perhaps nothing refers to it. Likewise, a row of a dependent table is not always a dependent row—the foreign key can allow null values, which refer to no other rows.

Referential constraints are optional. You define referential constraints by using CREATE TABLE and ALTER TABLE statements.

DB2 enforces referential constraints when the following actions occur:

- An INSERT statement is applied to a dependent table.
- An UPDATE statement is applied to a foreign key of a dependent table or to the parent key of a parent table.
- A MERGE statement that includes an insert operation is applied to a dependent table.
- A MERGE statement that includes an update operation is applied to a foreign key of a dependent table or to the parent key of a parent table.
- A DELETE statement is applied to a parent table. All affected referential constraints and all delete rules of all affected relationships must be satisfied in order for the delete operation to succeed.
- The LOAD utility with the ENFORCE CONSTRAINTS option is run on a dependent table.
- The CHECK DATA utility is run.

Another type of referential constraint is an *informational referential constraint*. This type of constraint is not enforced by DB2 during normal operations. An application process should verify the data in the referential integrity relationship. An informational referential constraint allows queries to take advantage of materialized query tables.

The order in which referential constraints are enforced is undefined. To ensure that the order does not affect the result of the operation, there are restrictions on the definition of delete rules and on the use of certain statements. The restrictions are specified in the descriptions of the SQL statements CREATE TABLE, ALTER TABLE, INSERT, UPDATE, MERGE, and DELETE.

The rules of referential integrity involve the following concepts and terminology:

parent key
A primary key or a unique key of a referential constraint.

parent table
A table that is a parent in at least one referential constraint. A table can be defined as a parent in an arbitrary number of referential constraints.

dependent table
A table that is a dependent in at least one referential constraint. A table can be defined as a dependent in an arbitrary number of referential constraints. A dependent table can also be a parent table.

descendent table
A table that is a dependent of another table or a table that is a dependent of a descendent table.

referential cycle
A set of referential constraints in which each associated table is a descendent of itself.

parent row
A row that has at least one dependent row.

dependent row
A row that has at least one parent row.

descendent row
A row that is dependent on another row or a row that is a dependent of a descendent row.

self-referencing row
A row that is a parent of itself.

self-referencing table
A table that is both parent and dependent in the same referential constraint. The constraint is called a *self-referencing constraint*.

The following rules provide referential integrity:

insert rule
A nonnull insert value of the foreign key must match some value of the parent key of the parent table. The value of a composite foreign key is null if any component of the value is null.

update rule
A nonnull update value of the foreign key must match some value of the parent key of the parent table. The value of a composite foreign key is treated as null if any component of the value is null.

delete rule
Controls what happens when a row of the parent table is deleted. The choices of action, made when the referential constraint is defined, are RESTRICT, NO ACTION, CASCADE, or SET NULL. SET NULL can be specified only if some column of the foreign key allows null values.

More precisely, the delete rule applies when a row of the parent table is the object of a delete or propagated delete operation and that row has dependents in the dependent table of the referential constraint. A *propagated delete* refers to the situation where dependent rows are deleted when parent rows are deleted. Let *P* denote the parent table, let *D* denote the dependent table, and let *p* denote a parent row that is the object of a delete or propagated delete operation. If the delete rule is:

- RESTRICT or NO ACTION, an error occurs and no rows are deleted.
- CASCADE, the delete operation is propagated to the dependent rows of *p* in *D*.
- SET NULL, each nullable column of the foreign key of each dependent row of *p* in *D* is set to null.

Each referential constraint in which a table is a parent has its own delete rule, and all applicable delete rules are used to determine the result of a delete operation. Thus, a row cannot be deleted if it has dependents in a referential constraint with a delete rule of RESTRICT or NO ACTION or the deletion cascades to any of its descendents that are dependents in a referential constraint with the delete rule of RESTRICT or NO ACTION.

The deletion of a row from parent table *P* involves other tables and can affect rows of these tables:

- If *D* is a dependent of *P* and the delete rule is RESTRICT or NO ACTION, *D* is involved in the operation but is not affected by the operation and the deletion from the parent table *P* does not take place.
- If *D* is a dependent of *P* and the delete rule is SET NULL, *D* is involved in the operation and rows of *D* might be updated during the operation.
- If *D* is a dependent of *P* and the delete rule is CASCADE, *D* is involved in the operation and rows of *D* might be deleted during the operation. If rows of *D* are deleted, the delete operation on *P* is said to be propagated to *D*. If *D* is also a parent table, the actions described in this list apply, in turn, to the dependents of *D*.

Any table that can be involved in a delete operation on *P* is said to be *delete-connected* to *P*. Thus, a table is delete-connected to table *P* if it is a dependent of *P* or a dependent of a table to which delete operations from *P* cascade.

 Department table (DSN8B10.DEPT) (Introduction to DB2 for z/OS)

 Employee table (DSN8B10.EMP) (Introduction to DB2 for z/OS)

 Project activity table (DSN8B10.PROJACT) (Introduction to DB2 for z/OS)

 Referential constraints (DB2 Application programming and SQL)

Check constraints

A *check constraint* is a rule that specifies the values that are allowed in one or more columns of every row of a base table.

Like referential constraints, check constraints are optional and you define them by using the CREATE TABLE and ALTER TABLE statements. The definition of a check constraint restricts the values that a specific column of a base table can contain.

A table can have any number of check constraints. DB2 enforces a check constraint by applying the restriction to each row that is inserted, loaded, or updated. One restriction is that a column name in a check constraint on a table must identify a column of that table.

Example: You can create a check constraint to ensure that all employees earn a salary of $30 000 or more:

```
CHECK (SALARY>= 30000)
```

Related concepts:

Check constraints (DB2 Application programming and SQL)

Triggers

A *trigger* defines a set of actions that are executed when a delete, insert, or update operation occurs on a specified table or view. When an SQL operation is executed, the trigger is *activated*. You can use triggers with referential constraints and check constraints to enforce data integrity rules.

When an insert, load, update, or delete is executed, the trigger is activated.

You can use triggers along with referential constraints and check constraints to enforce data integrity rules. Triggers are more powerful than constraints because you can use them to do the following things:

- Update other tables
- Automatically generate or transform values for inserted or updated rows
- Invoke functions that perform operations both inside and outside of DB2

For example, assume that you need to prevent an update to a column when a new value exceeds a certain amount. Instead of preventing the update, you can use a trigger. The trigger can substitute a valid value and invoke a procedure that sends a notice to an administrator about the attempted invalid update.

You can define triggers with the CREATE TRIGGER statement.

INSTEAD OF triggers are triggers that execute instead of the INSERT, UPDATE, or DELETE statement that activates the trigger. Unlike other triggers, which are defined on tables only, INSTEAD OF triggers are defined on views only. INSTEAD OF triggers are particularly useful when the triggered actions for INSERT, UPDATE, or DELETE statements on views need to be different from the actions for SELECT statements. For example, an INSTEAD OF trigger can be used to facilitate an update through a join query or to encode or decode data in a view.

Triggers move the business rule application logic into the database, which results in faster application development and easier maintenance. The business rule is no longer repeated in several applications, and the rule is centralized to the trigger. DB2 checks the validity of the changes that any application makes to the salary column, and you are not required to change application programs when the logic changes.

Creation of triggers (Introduction to DB2 for z/OS)

Application processes, concurrency, and recovery

All SQL programs execute as part of an *application process*. An application process involves the execution of one or more programs, and it is the unit to which DB2 allocates resources and locks.

Different application processes might involve the execution of different programs, or different executions of the same program. The means of initiating and terminating an application process are dependent on the environment.

Locking, commit, and rollback

More than one application process might request access to the same data at the same time. Furthermore, under certain circumstances, an SQL statement can

execute concurrently with a utility on the same table space. *Locking* is used to maintain data integrity under such conditions, preventing, for example, two application processes from updating the same row of data simultaneously.

DB2 implicitly acquires locks to prevent uncommitted changes made by one application process from being perceived by any other. DB2 will implicitly release all locks it has acquired on behalf of an application process when that process ends, but an application process can also explicitly request that locks be released sooner. A *commit* operation releases locks acquired by the application process and commits database changes made by the same process.

DB2 provides a way to *back out* uncommitted changes made by an application process. This might be necessary in the event of a failure on the part of an application process, or in a *deadlock* situation. An application process, however, can explicitly request that its database changes be backed out. This operation is called *rollback*.

The interface used by an SQL program to explicitly specify these commit and rollback operations depends on the environment. If the environment can include recoverable resources other than DB2 databases, the SQL COMMIT and ROLLBACK statements cannot be used. Thus, these statements cannot be used in an IMS, CICS, or WebSphere® environment.

Unit of work

A *unit of work* is a recoverable sequence of operations within an application process. A unit of work is sometimes called a *logical unit of work*.

At any time, an application process has a single unit of work, but the life of an application process can involve many units of work as a result of commit or full rollback operations.

A unit of work is initiated when an application process is initiated. A unit of work is also initiated when the previous unit of work is ended by something other than the end of the application process. A unit of work is ended by a commit operation, a full rollback operation, or the end of an application process. A commit or rollback operation affects only the database changes made within the unit of work it ends. While these changes remain uncommitted, other application processes are unable to perceive them unless they are running with an isolation level of uncommitted read. The changes can still be backed out. Once committed, these database changes are accessible by other application processes and can no longer be backed out by a rollback. Locks acquired by DB2 on behalf of an application process that protects uncommitted data are held at least until the end of a unit of work.

The initiation and termination of a unit of work define *points of consistency* within an application process. A point of consistency is a claim by the application that the data is consistent. For example, a banking transaction might involve the transfer of funds from one account to another. Such a transaction would require that these funds be subtracted from the first account, and added to the second. Following the subtraction step, the data is inconsistent. Only after the funds have been added to the second account is consistency reestablished. When both steps are complete, the commit operation can be used to end the unit of work, thereby making the changes available to other application processes. The following figure illustrates this concept.

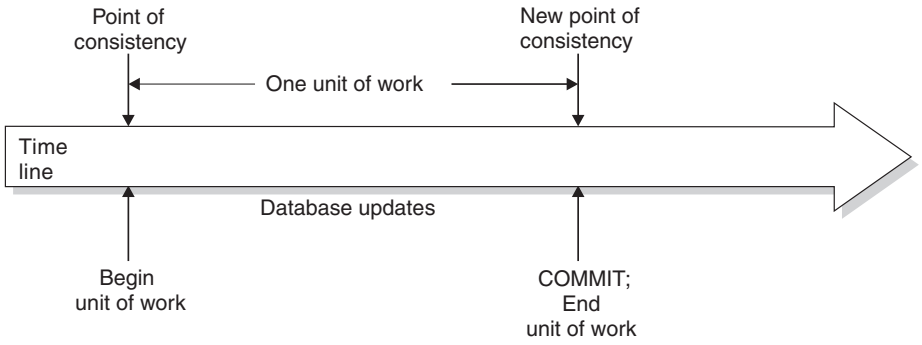


Figure 7. Unit of work with a commit operation

Unit of recovery

A *DB2 unit of recovery* is a recoverable sequence of operations executed by DB2 for an application process.

If a unit of work involves changes to other recoverable resources, the unit of work will be supported by other units of recovery. If relational databases are the only recoverable resources used by the application process, then the scope of the unit of work and the unit of recovery are the same and either term can be used.

Rolling back work

DB2 can back out all changes made in a unit of recovery or only selected changes. Only backing out all changes results in a point of consistency.

Rolling back all changes

The SQL ROLLBACK statement without the TO SAVEPOINT clause specified causes a full rollback operation. If such a rollback operation is successfully executed, DB2 backs out uncommitted changes to restore the data consistency that existed when the unit of work was initiated.

That is, DB2 undoes the work, as shown in the following figure:

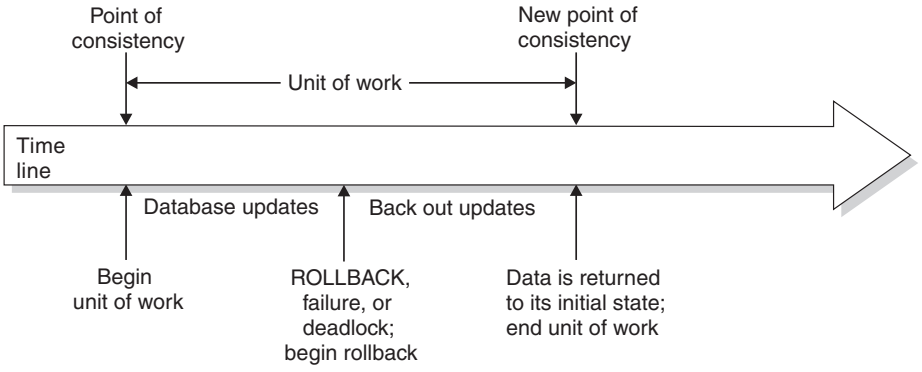


Figure 8. Rolling back all changes from a unit of work

Rolling back selected changes using savepoints

A *savepoint* represents the state of data at some particular time during a unit of work. An application process can set savepoints within a unit of work, and then as logic dictates, roll back only the changes that were made after a savepoint was set.

For example, part of a reservation transaction might involve booking an airline flight and then a hotel room. If a flight gets reserved but a hotel room cannot be reserved, the application process might want to undo the flight reservation without undoing any database changes made in the transaction prior to making the flight reservation. SQL programs can use the SQL SAVEPOINT statement to set savepoints, the SQL ROLLBACK statement with the TO SAVEPOINT clause to undo changes to a specific savepoint or the last savepoint that was set, and the SQL RELEASE SAVEPOINT statement to delete a savepoint. The following figure illustrates this concept.

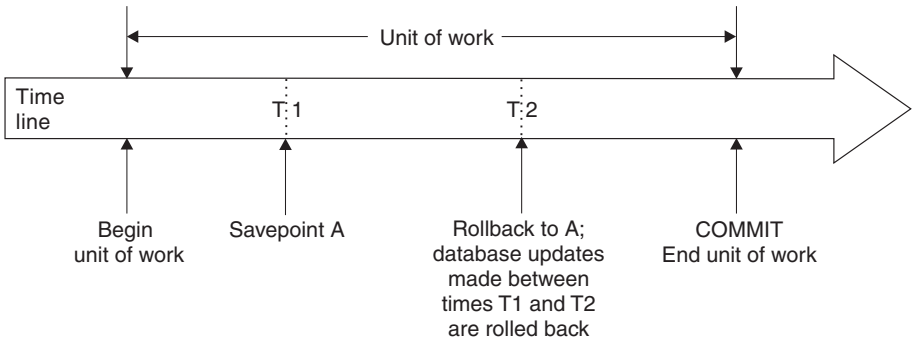


Figure 9. Rolling back changes to a savepoint within a unit of work

Packages and application plans

A *package* contains control structures that DB2 uses when it runs SQL statements. An *application plan* relates an application process to a local instance of DB2 and specifies processing options.

Packages are produced during program preparation. You can think of the control structures as the bound or operational form of SQL statements. All control structures in a package are derived from the SQL statements that are embedded in a single source program.

An application plan contains one or both of the following elements:
- A list of package names

DB2 applications require an application plan. Packages make application programs more flexible and easier to maintain.

Example: The following figure shows an application plan that contains two packages. Suppose that you decide to change the SELECT statement in package AA to select data from a different table. In this case, you need to bind only package AA again and not package AB. GUPI

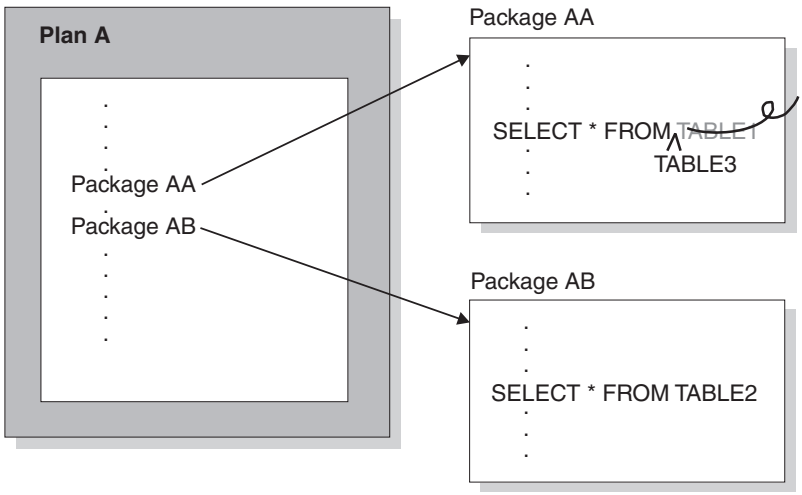


Figure 10. Application plan and packages

GUPI

In general, you create plans and packages by using the DB2 commands BIND PLAN and BIND PACKAGE.

A *trigger package* is a special type of package that is created when you execute a CREATE TRIGGER statement. A trigger package executes only when the trigger with which it is associated is activated.

Packages for JDBC, SQLJ, and ODBC applications serve different purposes that you can read more about later in this information.

Application programming for DB2 (Introduction to DB2 for z/OS)

Preparation process for an application program (Introduction to DB2 for z/OS)

"CREATE PROCEDURE (SQL - native)" on page 1350

"CREATE TRIGGER" on page 1482

"SET CURRENT PACKAGE PATH" on page 1899

"SET CURRENT PACKAGESET" on page 1903

Routines

A *routine* is an executable SQL object. The two types of routines are functions and stored procedures.

Functions

A *function* is a routine that can be invoked from within other SQL statements and that returns a value or a table.

Functions are classified as either SQL functions or external functions. SQL functions are written using SQL statements, which are also known collectively as SQL procedural language. External functions reference a host language program. The host language program can contain SQL, but does not require SQL.

You define functions by using the CREATE FUNCTION statement. You can classify functions as built-in functions, user-defined functions, or cast functions that are generated for distinct types. Functions can also be classified as aggregate, scalar, or table functions, depending on the input data values and result values.

A *table function* can be used only in the FROM clause of a statement. Table functions return columns of a table and resemble a table that is created through a CREATE TABLE statement. Table functions can be qualified with a schema name.

Creation of user-defined functions (Introduction to DB2 for z/OS)

Chapter 3, "Functions," on page 337

Stored procedures

A *procedure*, also known as a stored procedure, is a routine that you can call to perform operations that can include SQL statements.

Procedures are classified as either SQL procedures or external procedures. SQL procedures contain only SQL statements. External procedures reference a host language program that might or might not contain SQL statements.

DB2 for z/OS supports the following types of SQL procedures:

External stored procedures
External stored procedures are procedures that are written in a host language and can contain SQL statements. The source code for an external stored procedure is separate from the definition. You can write an external stored procedure in Assembler, C, C++, COBOL, Java, REXX, or PL/I. All programs must be designed to run using Language Environment®. Your COBOL and C++ stored procedures can contain object-oriented extensions.

External SQL procedures
External SQL procedures are procedures whose body is written in SQL. DB2 supports them by generating an associated C program for each procedure. All SQL procedures that were created prior to Version 9.1 are external SQL procedures. Starting in Version 9.1, you can create an external SQL procedure by specifying FENCED or EXTERNAL in the CREATE PROCEDURE statement.

Native SQL procedures
Native SQL procedures are procedures whose body is written in SQL. For native SQL procedures, DB2 does not generate an associated C program. Starting in Version 9.1, all SQL procedures that are created without the FENCED or EXTERNAL options in the CREATE PROCEDURE statement are native SQL procedures. You can create native SQL procedures in one step. Native SQL statements support more functions and usually provide better performance than external SQL statements.

SQL control statements are supported in SQL procedures. Control statements are SQL statements that allow SQL to be used in a manner similar to writing a program in a structured programming language. SQL control statements provide the capability to control the logic flow, declare and set variables, and handle warnings and exceptions. Some SQL control statements include other nested SQL statements.

SQL procedures provide the same benefits as procedures in a host language. That is, a common piece of code needs to be written and maintained only once and can be called from several programs.

SQL procedures provide additional benefits when they contain SQL statements. In this case, SQL procedures can reduce or eliminate network delays that are associated with communication between the client and server and between each SQL statement. SQL procedures can improve security by providing a user the ability to invoke only a procedure instead of providing them with the ability to execute the SQL that the procedure contains.

You define procedures by using the CREATE PROCEDURE statement.

Use of an application program as a stored procedure (Introduction to DB2 for z/OS)

External stored procedures (DB2 Application programming and SQL)

"SQL control statements for external SQL procedures" on page 2032

Chapter 6, "SQL control statements for SQL routines," on page 1963

Sequences

A *sequence* is a stored object that simply generates a sequence of numbers in a monotonically ascending (or descending) order. A sequence provides a way to have DB2 automatically generate unique integer primary keys and to coordinate keys across multiple rows and tables.

A sequence can be used to exploit parallelization, instead of programmatically generating unique numbers by locking the most recently used value and then incrementing it.

Sequences are ideally suited to the task of generating unique key values. One sequence can be used for many tables, or a separate sequence can be created for each table requiring generated keys. A sequence has the following properties:

- Guaranteed, unique values, assuming that the sequence is not reset and does not allow the values to cycle
- Monotonically increasing or decreasing values within a defined range
- Can increment with a value other than 1 between consecutive values (the default is 1).
- Recoverable. If DB2 should fail, the sequence is reconstructed from the logs so that DB2 guarantees that unique sequence values continue to be generated across a DB2 failure.

Values for a given sequence are automatically generated by DB2. Use of DB2 sequences avoids the performance bottleneck that results when an application implements sequences outside the database. The counter for the sequence is incremented (or decremented) independently of the transaction. In some cases, gaps can be introduced in a sequence. A gap can occur when a given transaction increments a sequence two times. The transaction might see a gap in the two numbers that are generated because there can be other transactions concurrently incrementing the same sequence. A user might not realize that other users are drawing from the same sequence. Furthermore, it is possible that a given sequence can appear to have generated gaps in the numbers, because a transaction that might have generated a sequence number might have rolled back or the DB2 subsystem might have failed. Updating a sequence is not part of a transaction's unit of recovery.

A sequence is created with a CREATE SEQUENCE statement. A sequence can be referenced using a *sequence-reference*. A sequence reference can appear most places

that an expression can appear. A sequence reference can specify whether the value to be returned is a newly generated value, or the previously generated value.

Although there are similarities, a sequence is different than an identity column. A sequence is an object, whereas an identity column is a part of a table. A sequence can be used with multiple tables, but an identity column is tied to a single table.

"CREATE SEQUENCE" on page 1375

User-defined types

A *user-defined data type* is a data type that is defined to the database using a CREATE statement.

A user-defined data types is a distinct type or an array type.

A *distinct type* is a user-defined type that shares its internal representation with a built-in data type (its source type), but is considered to be a separate and incompatible data type for most operations. A distinct type is created with an SQL CREATE TYPE (distinct) statement. A distinct type can be used to define a column of a table, or a parameter of a routine.

An *array type* is a user-defined data type that consists of an ordered set of elements of a single built-in data type. Elements can be accessed and modified by their index position. An array type is created with an SQL CREATE TYPE (array) statement. An array type can be used as a parameter of a procedure and as a variable in an SQL procedure.

Related concepts:

"Array types" on page 108

"Distinct types" on page 107

Related reference:

"CREATE TYPE" on page 1510

Distributed data

The database managers in a distributed relational database communicate and cooperate with each other in a way that allows a DB2 application program to use SQL to access data at any of the interconnected computer systems.

A *distributed relational database* consists of a set of tables and other objects that are spread across different, but interconnected, computer systems. Each computer system has a relational database manager, such as DB2, that manages the tables in its environment. The database managers communicate and cooperate with each other in a way that allows a DB2 application program to use SQL to access data at any of the computer systems. The DB2 subsystem where the application plan is bound is known as the *local DB2 subsystem*. Any database server other than the local DB2 subsystem is considered a *remote database server*, and access to its data is a distributed operation.

Distributed relational databases are built on formal requester-server protocols and functions. An *application requester* component supports the application end of a connection. It transforms an application's database request into communication protocols that are suitable for use in the distributed database network. These requests are received and processed by an *application server* component at the database server end of the connection. Working together, the application requester

and application server handle the communication and location considerations so that the application is isolated from these considerations and can operate as if it were accessing a local database.

For more information on Distributed Relational Database Architecture™ (DRDA®) communication protocols, see *Open Group Technical Standard, DRDA Version 3 Vol. 1: Distributed Relational Database Architecture*.

Related concepts:

Distributed data access (Introduction to DB2 for z/OS)

Related tasks:

Improving performance for applications that access distributed data (DB2 Performance)

Connections

A *connection* is an association between an application process and a local or remote database server. Connections are managed by applications.

An application process must be connected to the application server facility of a database manager before SQL statements that reference tables or views can be executed. An application can use the CONNECT statement to establish a connection to a database server and make that database server the current server of the application process.

Commit processing: When DB2 for z/OS acts as a requester, it negotiates with the database server during the connection process to determine how to perform commits. If the remote server does not support two-phase commit protocol, DB2 downgrades to perform one-phase commits. Otherwise, DB2 always performs two-phase commits, which allow applications to update one or more databases in a single unit of work and are more reliable than one-phase commits. Two-phase commit is a two-step process:

1. First, all database managers involved in the same unit of work are pooled to determine whether they are ready to commit.
2. Then, if all database managers respond positively, they are directed to execute commit processing. If all database managers do not respond positively, they are directed to execute backout processing.

DB2 can also provide coordination for transactions that include both two-phase commit resources and one-phase commit resources. If an application has multiple connections to several different database servers, and if any of the connections are one-phase commit connections, then only one database that is involved in the transaction can be updated. The connections to all the other databases that are involved in the transaction are read-only.

Supported SQL statements and clauses: For the most part, an application can use the statements and clauses that are supported by the database manager of the current server, even though that application might be running via the application requester of a database manager that does not support some of those statements and clauses. Restrictions to this general rule for DB2 for z/OS are documented in *IBM DB2 SQL Reference for Cross-Platform Development*.

To execute a static SQL statement that references tables or views, the bound form of the statement is taken from a package that the database manager previously created through a bind operation or when a version of a native SQL procedure was defined.

Distributed unit of work

The *distributed unit of work facility* provides for the remote preparation and execution of SQL statements.

An application process at computer system A can connect to a database server at computer system B and, within one or more units of work, execute any number of static or dynamic SQL statements that reference objects at B. All objects referenced in a single SQL statement must be managed by the same database server. Any number of database servers can participate in the same unit of work, and any number of connections can exist between an application process and a database server. A commit or rollback operation that does not specify a savepoint ends the unit of work.

Connection management

How connections are managed depends on what states the SQL connection and the application process are in.

At any time:

- An SQL connection is in one of the following states:
  - Current and held
  - Current and release-pending
  - Dormant and held
  - Dormant and release-pending
- An application process is in the connected or unconnected state, and has a set of zero or more SQL connections. Each SQL connection is uniquely identified by the name of the database server at the other end of the connection. Only one SQL connection is active (current) at a time.

Initial state of an application process: An application process is initially in the connected state, and it has exactly one SQL connection. The server of that connection is the local DB2 subsystem.

Initial state of an SQL connection: An SQL connection is initially in the current and held state.

The following figure shows the state transitions:

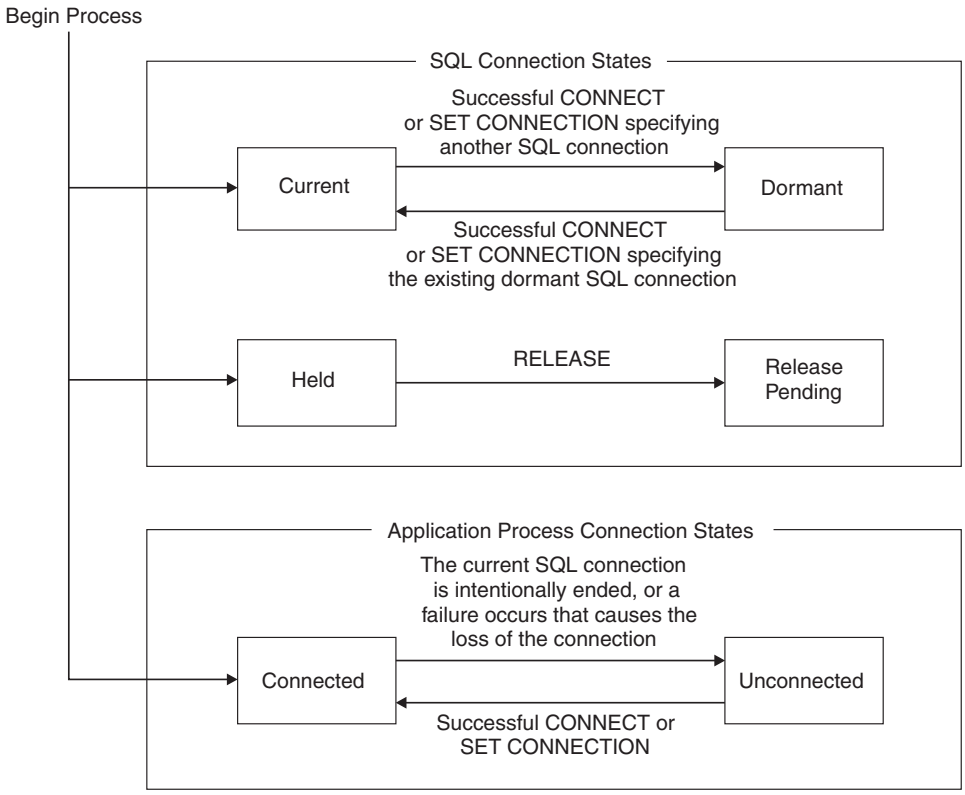


Figure 11. State transitions for an SQL connection and an application process connection in a distributed unit of work

SQL connection states

If an application process successfully executes a CONNECT statement, the SQL connection states of the connections change.

If an application process successfully executes a CONNECT statement:
- The current connection is placed in the dormant and held state.
- The new connection is placed in the current and held state.
- The location name is added to the set of existing connections.

If the location name is already in the set of existing connections, an error is returned.

An SQL connection in the dormant state is placed in the current state using:
- The SET CONNECTION statement, or
- The CONNECT statement, if the SQLRULES(DB2) bind option is in effect.

When an SQL connection is placed in the current state, the previously-current SQL connection, if any, is placed in the dormant state. No more than one SQL connection in the set of existing connections of an application process can be current at any time. Changing the state of an SQL connection from current to dormant or from dormant to current has no effect on its held or release-pending state.

An SQL connection is placed in the release-pending state by the RELEASE statement. When an application process executes a commit operation, every

release-pending connection of the process is ended. Changing the state of an SQL connection from held to release-pending has no effect on its current or dormant state. Thus, an SQL connection in the release-pending state can still be used until the next commit operation. No way exists to change the state of a connection from release-pending to held.

Application process connection states

In a distributed unit of work, an application process can be in a connected or unconnected state. Depending on the state, the application process can execute only certain SQL statements successfully.

A connection to a different database server can be established by the explicit or implicit execution of a CONNECT statement. The following rules apply:

- An application process cannot have more than one SQL connection to the same database server at the same time.
- When an application process executes a SET CONNECTION statement, the specified location name must be in the set of existing connections of the application process.
- When an application process executes a CONNECT statement and the SQLRULES(STD) bind option is in effect, the specified location must not be in the set of existing connections of the application process.

If an application process has a current SQL connection, the application process is in a *connected* state. The CURRENT SERVER special register contains the name of the database server of the current SQL connection. The application process can execute SQL statements that refer to objects managed by that server. If the server is a DB2 subsystem, the application process can also execute certain SQL statements that refer to objects managed by a DB2 subsystem with which that server can establish a connection.

An application process in an unconnected state enters a connected state when it successfully executes a CONNECT or SET CONNECTION statement.

If an application process does not have a current SQL connection, the application process is in an *unconnected* state. The CURRENT SERVER special register contains blanks. The only SQL statements that can be executed successfully are CONNECT, RELEASE, COMMIT, ROLLBACK, and any of the following local SET statements.

- SET CONNECTION
- SET CURRENT APPLICATION ENCODING SCHEME
- SET CURRENT PACKAGE PATH
- SET CURRENT PACKAGESET
- SET *host-variable* = CURRENT APPLICATION ENCODING SCHEME
- SET *host-variable* = CURRENT PACKAGESET
- SET *host-variable* = CURRENT SERVER

Because the application process is in an unconnected state, a COMMIT or ROLLBACK statement is processed by the local DB2 subsystem.

An application process in a connected state enters an unconnected state when its current SQL connection is intentionally ended, or the execution of an SQL statement is unsuccessful because of a failure that causes a rollback operation at the current server and loss of the SQL connection. SQL connections are intentionally ended when an application process successfully executes a commit operation and either of the following are true:

- The SQL connection is in the release-pending state.
- The SQL connection is not in the release-pending state, but it is a remote connection and either of the following are true:
  - The DISCONNECT(AUTOMATIC) bind option is in effect
  - The DISCONNECT(CONDITIONAL) bind option is in effect and an open WITH HOLD cursor is not associated with the connection

An implicit CONNECT to a default DB2 subsystem is executed when an application process executes an SQL statement other than COMMIT, CONNECT TO, CONNECT RESET, SET CONNECTION, RELEASE, or ROLLBACK, and if all of the following conditions apply:

- The CURRENTSERVER bind option was specified when creating the application plan of the application process and the identified server is not the local DB2.
- An explicit CONNECT statement has not already been successfully or unsuccessfully executed by the application process.
- An implicit connection has not already been successfully or unsuccessfully executed by the application process. An implicit connection occurs as the result of execution of an SQL statement that contains a three-part name in a package that is bound with the DBPROTOCOL(DRDA) option.

If the implicit CONNECT fails, the application process is placed in an *unconnected* state.

When a connection is ended, all resources that were acquired by the application process through the connection and all resources that were used to create and maintain the connection are returned to the connection pool. For example, if application process *P* placed the connection to application server *X* in the release-pending state, all cursors of *P* at *X* are closed and returned to the connection pool when the connection is ended during the next commit operation.

When a connection is ended as a result of a communications failure, the application process is placed in an unconnected state.

All connections of an application process are ended when the process ends.

Remote unit of work

The *remote unit of work facility* also provides for the remote preparation and execution of SQL statements, but in a much more restricted fashion than the distributed unit of work facility.

An application process at computer system A can connect to a database server at computer system B and, within one or more units of work, execute any number of static or dynamic SQL statements that reference objects at B. All objects referenced in a single SQL statement must be managed by the same database server, and all SQL statements in the same unit of work must be executed by the same database server. However, unlike a distributed unit of work, an application process can have only one connection at a time. The process cannot connect to a new server until it executes a commit or rollback operation on the current server to end that unit of work. This restricts the situations in which a CONNECT statement can be executed.

Connection management

How connections are managed depends on what states the SQL connection and the application process are in.

An application process is in one of four states at any time:
- Connectable and connected
- Unconnectable and connected
- Connectable and unconnected
- Unconnectable and unconnected

Initial state of an application process: An application process is initially in the connectable and connected state. The database server to which the application process is connected is determined by a product-specific option that might involve an implicit CONNECT operation. An implicit connect operation cannot occur if an implicit or explicit connect operation has already successfully or unsuccessfully occurred. Thus, an application process cannot be implicitly connected to a database server more than once. Other rules for implicit connect operations are product-specific.

Figure 12 shows the state transitions:

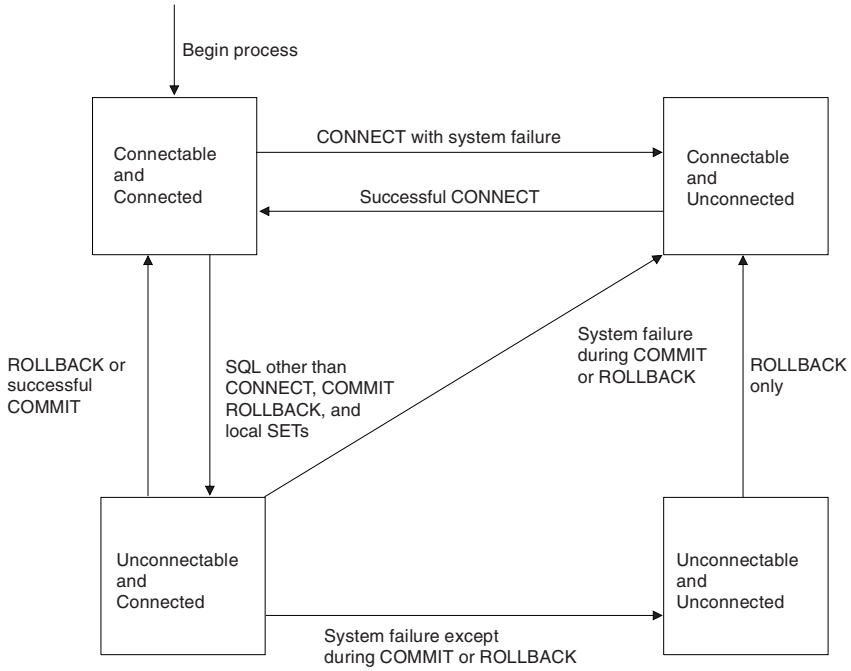


Figure 12. State transitions for an application process connection in a remote unit of work

In the following descriptions of application process connections, CONNECT can mean:
- CONNECT TO
- CONNECT RESET
- CONNECT *authorization*

It cannot mean CONNECT with no operand, which is used to return information about the current server.

Consecutive CONNECT statements can be executed successfully because CONNECT does not remove an application process from the connectable state. A CONNECT statement does not initiate a new unit of work; a unit of work is initiated by the first SQL statement that accesses data. CONNECT cannot execute successfully when it is preceded by any SQL statement other than CONNECT,

COMMIT, RELEASE, ROLLBACK, or SET CONNECTION. To avoid an error, execute a commit or rollback operation before a CONNECT statement is executed.

Connectable and connected state: In the connectable and connected state, an application process is connected to a database server, and CONNECT statements that target the current server can be executed. An application process re-enters this state when either of the following is true:

- The process completes a rollback or a successful commit from an unconnectable and connected state.
- The process successfully executes a CONNECT statement from a connectable and unconnected state.

Unconnectable and connected state: In the unconnectable and connected state, an application process is connected to a database server, and only a CONNECT statement with no operands can be executed. An application process enters this state from a connectable and connected state when it executes any SQL statement other than CONNECT, COMMIT, or ROLLBACK.

Connectable and unconnected state: In the connectable and unconnected state, an application process is not connected to a database server. The only SQL statement that can be executed is CONNECT. An application process enters this state if any of the following is true:

- The process does not successfully execute a CONNECT statement from a connectable and connected state.
- The process executes a COMMIT statement when the SQL connection is in a release-pending state.
- A system failure occurs during a COMMIT or ROLLBACK from an unconnectable and connected state.
- The process executes a ROLLBACK statement from an unconnectable and unconnected state.

Other product-specific reasons can also cause an application process to enter the connectable and unconnected state.

Unconnectable and unconnected state: In the unconnectable and unconnected state, an application process is not connected to a database server and CONNECT statements cannot be executed. The only SQL statement that can be executed is ROLLBACK. An application process enters this state from an unconnectable and connected state as a result of a system failure, except during a COMMIT or ROLLBACK, at the server.

Character conversion

A *string* is a sequence of bytes that can represent characters. Within a string, all the characters are represented by a common encoding representation. In some cases, it might be necessary to convert these characters to a different encoding representation. The process of conversion is known as *character conversion*.

Character conversion, when required, is automatic, and when successful, it is transparent to the application.

In client/server environments, character conversion can occur when an SQL statement is executed remotely. Consider, for example, the following two cases. In either case, the data could have a different representation at the sending and receiving systems.

- The values of data sent from the requester to the current server
- The values of data sent from the current server to the requester

Conversion can also occur during string operations on the same system, as in the following examples:

- An overriding CCSID is specified.

  For example, an SQL statement with a descriptor, which requires an SQLDA. In the SQLDA, the CCSID is in the SQLNAME field for languages other than REXX, and in the SQLCCSID field for REXX. (For more information, see "SQL descriptor area (SQLDA)" on page 2079). A DECLARE VARIABLE statement can also be issued to associate a CCSID with the host variables into which data is retrieved from a table.
- The value of the ENCODING bind option or the APPLICATION ENCODING SCHEMA option of the CREATE PROCEDURE or ALTER PROCEDURE statement for a native SQL procedure (static SQL statements) or the CURRENT APPLICATION ENCODING SCHEME special register (for dynamic SQL) is different than encoding scheme of the data being retrieved.
- A mixed character string is assigned to an SBCS column or host variable.
- An SQL statement refers to data that is defined with different CCSIDs.

The text of an SQL statement is also subject to character conversion because it is a character string.

The following list defines some of the terms used for character conversion.

ASCII Acronym for American Standard Code for Information Interchange, an encoding scheme used to represent characters. The term ASCII is used throughout this information to refer to IBM-PC Data or ISO 8-bit data.

character set
A defined set of characters, a character being the smallest component of written language that has semantic value. For example, the following character set appears in several code pages:

- 26 nonaccented letters A through Z
- 26 nonaccented letters a through z
- digits 0 through 9
- . , : ; ? ( ) ' " / - _ & + % * = < >

code page
A set of assignments of characters to code points. For example, in EBCDIC, "A" is assigned code point X'C1', and "B" is assigned code point X'C2'. In Unicode UTF-8, "A" is assigned code point X'41', and "B" is assigned code point X'42'. Within a code page, each code point has only one specific meaning.

code point
A unique bit pattern that represents a character. It is a numerical index or position in an encoding table used for encoding characters.

coded character set
A set of unambiguous rules that establishes a character set and the

one-to-one relationships between the characters of the set and their coded representations. It is the assignment of each character in a character set to a unique numeric code value.

coded character set identifier (CCSID)
A two-byte, unsigned binary integer that uniquely identifies an encoding scheme and one or more pairs of character sets and code pages.

EBCDIC
Acronym for Extended Binary-Coded Decimal Interchange Code, an encoding scheme used to represent character data, a group of coded character sets that consist of 8 bit coded characters. EBCDIC coded character sets use the first 64 code positions (X'00' to X'3F') for control codes. The range X'41' to X'FE' is used for single-byte characters. For double-byte characters, the first byte is in the range X'41' to X'FE' and the second byte is also in the range X'41' to X'FE', while X'4040' represents a double-byte space.

encoding scheme
A set of rules used to represent character data. All string data stored in a table must use the same encoding scheme and all tables within a table space must use the same encoding scheme, except for global temporary tables, declared temporary tables, and work file table spaces. DB2 supports these encoding schemes:
- ASCII
- EBCDIC
- Unicode

substitution character
A unique character that is substituted during character conversion for any characters in the source encoding representation that do not have a match in the target encoding representation.

Unicode
A universal encoding scheme for written characters and text that enables the exchange of data internationally. It provides a character set standard that can be used all over the world. It provides the ability to encode all characters used for the written languages of the world and treats alphabetic characters, ideographic characters, and symbols equivalently because it specifies a numeric value and a name for each of its characters. It includes punctuation marks, mathematical symbols, technical symbols, geometric shapes, and dingbats. DB2 supports these two encoding forms:
- UTF-8: Unicode Transformation Format, a 8 bit encoding form designed for ease of use with existing ASCII-based systems. UTF-8 can encode any of the Unicode characters. A UTF-8 character is 1,2,3, or 4 bytes in length. A UTF-8 data string can contain any combination of SBCS and MBCS data, including supplementary characters. The CCSID value for data in UTF-8 format is 1208.
- UTF-16: Unicode Transformation Format, a 16 bit encoding form designed to provide code values for over a million characters and a superset of UCS-2. UTF-16 can encode any of the Unicode characters. In UTF-16 encoding, characters are 2 bytes in length, except for supplementary characters, which take two 2 byte string units per character. The CCSID value for data in UTF-16 format is 1200.

Character data (CHAR, VARCHAR, and CLOB) is encoded in Unicode UTF-8. Character strings are also used for mixed data (that is a mixture of single-byte characters and multi-byte characters) and for data that is not

associated with any character set (called bit data). Graphic data (GRAPHIC, VARGRAPHIC, and DBCLOB) is encoded in Unicode UTF-16. For a comparison of some UTF-8 and UTF-16 code points for some sample characters, see Figure 15 on page 46. This table shows how a UTF-8 character can be 1 to 4 bytes in length, a non-supplementary UTF-16 character is 2 bytes in length, and how a supplementary character in either UTF-8 or UTF-16 takes two 2 byte code points.

Character conversion can affect the results of several SQL operations. In this information, the effects are described in:

"Conversion rules for string assignment" on page 129
"Conversion rules for comparisons" on page 138
"Character conversion in set operations and concatenations" on page 816

Character sets and code pages

Even with the same encoding scheme, different CCSIDs exist, and the same code point can represent a different character in different CCSIDs. Furthermore, a byte in a character string does not necessarily represent a character from a single-byte character set (SBCS).

The following figure shows how a typical character set might map to different code points in two different code pages.

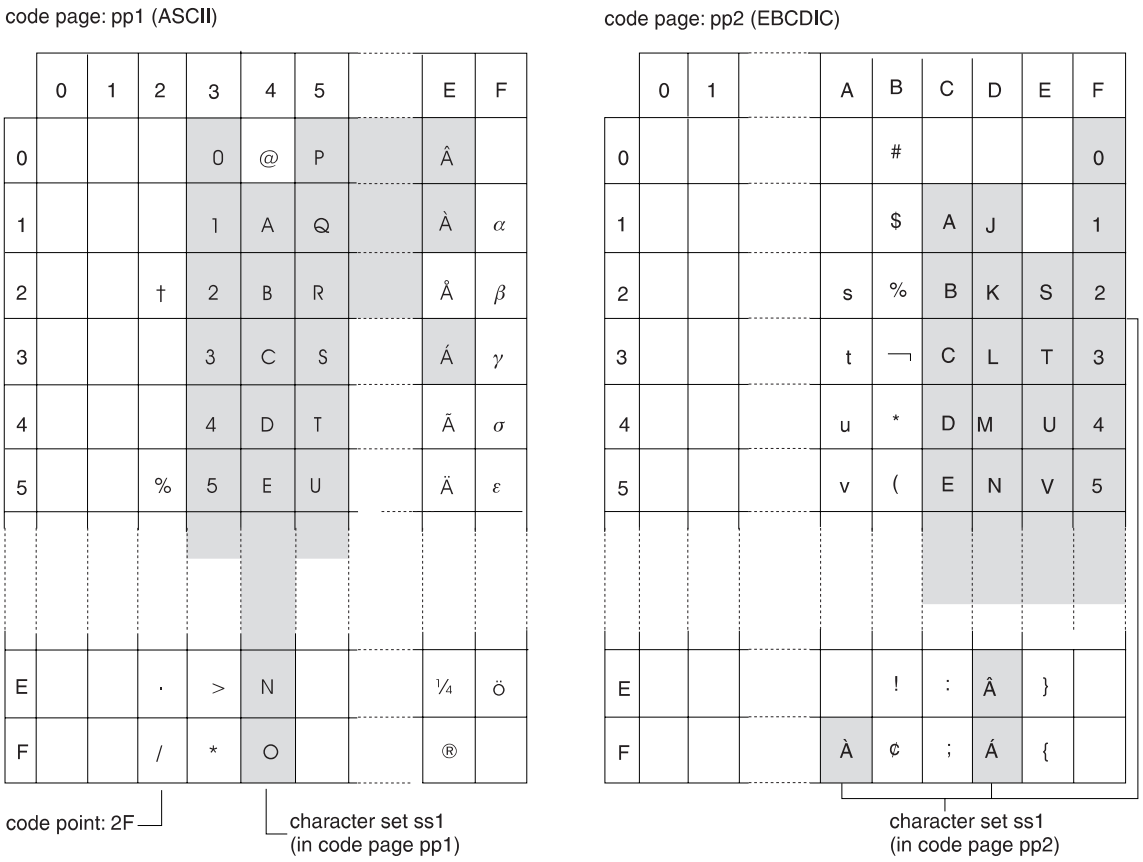


Figure 13. Code page mappings for character set ss1 in ASCII and EBCDIC

For Unicode, there is only one CCSID for UTF-8 and only one CCSID for UTF-16. The following figure shows how the first 127 single code points for UTF-8 are the same as ASCII with a CCSID of 367. For example, in both UTF-8 and ASCII CCSID

367, an A is X'41' and a 1 is X'31'.

First 127 code points for UTF-8 code page

	0	1	2	3	4	5	6	7
0			(SP)	0	@	P	`	p
1			!	1	A	Q	a	q
2			"	2	B	R	b	r
3			#	3	C	S	c	s
4			$	4	D	T	d	t
5			%	5	E	U	e	u
6			&	6	F	V	f	v
7			'	7	G	W	g	w
8			(	8	H	X	h	x
9			)	9	I	Y	i	y
A			*	:	J	Z	j	z
B			+	;	K	[	k	{
C			,	<	L	\	l	|
D			-	=	M	]	m	}
E			.	>	N	^	n	~
F			/	?	O	_	o	

code point: 2F

Figure 14. Code point mapping for the first 127 code points for UTF-8 single-byte characters (CCSID 1208)

The following figure shows a comparison of how some UTF-16 and UTF-8 code points map to some sample characters. The character for the eighth note musical symbol takes two 2 byte code points because it is a supplementary character.

Character glyph	UTF-8 code point	UTF-16 code point
M	4D	004D
Ä	C384	00C4
事	E4BA8B	4E8B
♪	F09D85A0	D834DD60

Figure 15. A comparison of how some UTF-8 and UTF-16 code points map to some sample characters

Related information:

Unicode Consortium

Character Data Representation Architecture Reference

Coded character sets and CCSIDS

IBM's character data representation architecture (CDRA) deals with the differences in string representation and encoding. The *Coded Character Set Identifier (CCSID)* is a key element of this architecture. A CCSID is a 2 byte (unsigned) binary number that uniquely identifies an encoding scheme and one or more pairs of character sets and code pages.

A CCSID is an attribute of strings, just as length is an attribute of strings. All values of the same string column have the same CCSID.

Character conversion is described in terms of CCSIDs of the source and target. With DB2 for z/OS, two methods are used to identify valid source and target combinations and to perform the conversion from one coded character set to another:

- DB2 catalog table SYSIBM.SYSSTRINGS

  Each row in the catalog table describes a conversion from one coded character set to another.
- z/OS support for Unicode

  For more information about the conversion services that are provided, including a complete list of the IBM-supplied conversion tables, see *z/OS Support for Unicode: Using Conversion Services*.

In some cases, no conversion is necessary even though the strings involved have different CCSIDs.

Different types of conversions might be supported by each database manager. Round-trip conversions attempt to preserve characters in one CCSID that are not defined in the target CCSID so that if the data is subsequently converted back to the original CCSID, the same original characters result. Enforced subset match conversions do not attempt to preserve such characters. Which type of conversion is used for a specific source and target CCSID is product-specific.

For more information on character conversion, see *DB2 Installation Guide*.

Determining the encoding scheme and CCSID of a string

An encoding scheme and a CCSID are attributes of strings, just as length is an attribute of strings. All values of the same string column have the same encoding scheme and CCSID.

Every string has an encoding scheme and a CCSID that identifies the manner in which the characters in the string are encoded. Strings can be encoded in ASCII, EBCDIC, or Unicode.

The CCSID that is associated with a string value depends on the SQL statement in which the data is referenced and the type of expression. Table 4 on page 48 describes the rules for determining the CCSID that is associated with a string value. Use the Type 1 rules when the SQL statement meets the following conditions:

- The SQL statement operates with a single set of CCSIDs (SBCS, mixed, and graphic). An SQL statement that does not contain any of the following items operates with a single set of CCSIDs:
  - References to columns from multiple tables or views that are defined with CCSIDs from more than one set of CCSIDs (SBCS, mixed, and graphic)
  - References to an EBCDIC table that contains a Unicode column
  - Graphic hexadecimal (GX) or hexadecimal Unicode (UX) string constants
  - References to the XMLCLOB built-in function
  - Cast specifications with a CCSID clause
  - User-defined table functions
- The SQL statement is not one of the following statements:
  - CALL statement
  - SET assignment statement
  - SET special register
  - VALUE statement
  - VALUES INTO statement
- One of the following built-in functions is not referenced:
  - XMLSERIALIZE
  - GETVARIABLE
  - DECRYPT_CHAR
  - DECRYPT_DB
  - DECRYPT_BIT
  - NORMALIZE_STRING
  - ASCII_CHR
  - CHAR
  - ASCII_STR (or ASCIISTR)
  - EBCDIC_CHR
  - EBCDIC_STR
  - UNICODE_STR (or UNISTR)
- The SQL statement does not include a collection-derived table (UNNEST).

Use the Type 2 rules when the statement does not meet the conditions for Type 1 rules.

Table 4. Rules for determining the CCSID that is associated with string data

Source of the string data	Type 1 rules	Type 2 rules
String constant	If the statement references a table or view, the encoding scheme of that table or view determines the encoding scheme for the string constant. Otherwise, the default EBCDIC encoding scheme is used for the string constant. The CCSID is the appropriate character string CCSID for the encoding scheme.	The CCSID is the appropriate character string CCSID for the application encoding scheme.[1]

Table 4. Rules for determining the CCSID that is associated with string data (continued)

Source of the string data	Type 1 rules	Type 2 rules
Datetime constant	If the statement references a table or view, the encoding scheme of that table or view determines the encoding scheme for the string constant. Otherwise, the default EBCDIC encoding scheme is used for the string constant. The CCSID is the appropriate character string CCSID for the encoding scheme.	The CCSID is the appropriate character string CCSID for the application encoding scheme.[1]
Hexadecimal string constant (X'...')	If the statement references a table or view, the encoding scheme of that table or view determines the encoding scheme for the string constant. Otherwise, the default EBCDIC encoding scheme is used for the string constant. The CCSID is the appropriate graphic string CCSID for the encoding scheme.	The CCSID is the appropriate character string CCSID for the application encoding scheme.[1]
Graphic string constant (G'...')	If the statement references a table or view, the encoding scheme of that table or view determines the encoding scheme for the graphic string constant. Otherwise, the default EBCDIC encoding scheme is used for the graphic string constant. The CCSID is the graphic string CCSID for the encoding scheme.	The CCSID is the graphic string CCSID for the application encoding scheme.[1]
Graphic hexadecimal constant (GX'...')	Not applicable.	The CCSID is the graphic string CCSID for the application encoding scheme, which must be ASCII or EBCDIC.
Hexadecimal Unicode string constant (UX'....')	Not applicable.	The CCSID is 1200 (UTF-16).
Special register	If the statement references a table or view, the encoding scheme of that table or view determines the encoding scheme for the special register. Otherwise, the default EBCDIC encoding scheme is used for the special register. The CCSID is the appropriate character string CCSID for the encoding scheme.	The CCSID is the appropriate CCSID for the application encoding scheme.[1]
Column of a table	The CCSID is the CCSID that is associated with the column of the table.	The CCSID is the CCSID that is associated with the column of the table.
Column of a view	The CCSID is the CCSID of the column of the result table of the fullselect of the view definition.	The CCSID is the CCSID of the column of the result table of the fullselect of the view definition.
Expression	The CCSID is the CCSID of the result of the expression.	The CCSID is the CCSID of the result of the expression.

Table 4. Rules for determining the CCSID that is associated with string data (continued)

Source of the string data	Type 1 rules	Type 2 rules
Result of a built-in function	If the description of the function, in Chapter 3, "Functions," on page 337, indicates what the CCSID of the result is, the CCSID is that CCSID. Otherwise, if the description of the function refers to this table for the CCSID, the CCSID is the appropriate CCSID of the CCSID set that is used by the statement for the data type of the result.	If the description of the function, in Chapter 3, "Functions," on page 337, indicates what the CCSID of the result is, the CCSID is that CCSID. Otherwise, if the description of the function refers to this table for the CCSID, the CCSID is the appropriate CCSID of the application encoding scheme for the data type of the result.[1]
Parameter of a user-defined routine	The CCSID is the CCSID that was determined when the function or procedure was created.	The CCSID is the CCSID that was determined when the function or procedure was created.
The expression in the RETURN statement of a CREATE statement for a user-defined SQL scalar function	If the expression in the RETURN statement is string data, the encoding scheme is the same as for the parameters of the function. The CCSID is determined from the encoding scheme and the attributes of the data.	The CCSID is determined from the CCSID of the result of the expression specified in the RETURN statement.
String host variable	If the statement references a table or view, the encoding scheme of that table or view determines the encoding scheme for the host variable. Graphic variables are an exception if the table or view is ECDIC or ASCII and the value of the MIXED DATA field on the DSNTIPF panel is NO. In this case, the Unicode encoding scheme is used for the host variable. Otherwise, the default EBCDIC encoding scheme is used for the host variable. The CCSID is the appropriate CCSID for the data type of the host variable.	At package prepare time, the CCSID is the appropriate CCSID for the data type of the host variable for the application encoding scheme. Graphic variables are an exception if the table or view is ECDIC or ASCII and the value of the MIXED DATA field on the DSNTIPF panel is NO. In this case, the Unicode encoding scheme is used for the host variable. At run time, the CCSID specified in the declare variable statement, or as an override in the SQLDA. Otherwise, the CCSID is the appropriate CCSID for the data type of the host variable for the application encoding scheme.

Note: If the context is within a check constraint or trigger package, the CCSID is the appropriate CCSID for Unicode instead of the application encoding scheme.

The following examples show how these rules are applied.

Example 1: Assume that the default encoding scheme for the installation is EBCDIC and that the installation does not support mixed and graphic data. The following statement conforms to the rules for Type 1 in Table 4 on page 48. Therefore, the X'40' is interpreted as EBCDIC SBCS data because the statement references a table that is in EBCDIC. The CCSID for X'40' is the default EBCDIC SBCS CCSID for the installation.

```
SELECT * FROM EBCDIC_TABLE WHERE COL1 = X'40';
```

the result of the query includes each row that has a value in column COL1 that is equal to a single EBCDIC blank.

Example 2: The following statement references data from two different tables that use different encoding schemes. This statement does not conform to the rules for Type 1 statements in Table 4 on page 48. Therefore, the rules for Type 2 statements are used. The CCSID for X'40' is dependent on the current application encoding scheme. Assuming that the current application encoding scheme is EBCDIC, X'40' represents a single EBCDIC blank.

```
SELECT * FROM EBCDIC_TABLE, UNICODE_TABLE WHERE COL1 = X'40';
```

as with Example 1, the result of the query includes each row that has a value in column COL1 that is equal to a single EBCDIC blank. If the current application encoding scheme were ASCII or Unicode, X'40' would represent something different and the results of the query would be different.

Expanding conversions

An *expanding conversion* occurs when the length of the converted string is greater than that of the source string.

For example, an expanding conversion occurs when an ASCII mixed data string that contains DBCS characters is converted to EBCDIC mixed data. To prevent the loss of data on expanding conversions, use a varying-length string variable with a maximum length that is sufficient to contain the expansion.

Expanding conversions also can occur when string data is converted to or from Unicode. It can also occur between UTF-8 and UTF-16, depending on the data being converted. UTF-8 uses 1, 2, 3, or 4 bytes per character. UTF-16 uses 2 bytes per character, except for supplementary characters, which use two 2 byte code points for each character. If UTF-8 were being converted to UTF-16, a 1 byte character would be expanded to 2 bytes.

Contracting conversions

A *contracting conversion* occurs when the length of the converted string is smaller than that of the source string.

For example, a contracting conversion occurs when an EBCDIC mixed data string that contains DBCS characters is converted to ASCII mixed data due to the removal of shift codes.

Contracting conversions also can occur when string data is converted to or from Unicode data. It can also occur between UTF-8 and UTF-16, depending on the data being converted.

Chapter 2. Language elements

An understanding of the basic syntax of SQL and language elements that are common to many SQL statements can be helpful in using SQL with DB2 for z/OS.

The following topics provide information about these language elements:

Characters

The basic symbols of keywords and operators in the SQL language are *characters* that are classified as letters, digits, or special characters.

- A *letter* is any of the 26 uppercase (A through Z) and 26 lowercase (a through z) letters of the English alphabet.[1]
- A *digit* is any one of the characters 0 through 9.
- A *special character* is any character other than a letter or a digit.

1. Letters also include three code points reserved as alphabetic extenders for national languages ($, #, and @ in the United States). These three code points (X'5B', X'7B', and X'7C') should be avoided because they represent different characters depending on the CCSID.

Tokens

The basic syntactical units of the SQL language are called *tokens*. A token consists of one or more characters of which none are blanks, control characters, or characters within a string constant or delimited identifier.

Tokens are classified as *ordinary* or *delimiter* tokens:

- An *ordinary token* is a numeric constant, an ordinary identifier, a host identifier, or a keyword.

  Examples:

  ```
  1       .1       +2       SELECT       E       3
  ```

- A *delimiter token* is a string constant, a delimited identifier, an operator symbol, or any of the special characters shown in the syntax diagrams. A question mark (?) is also a delimiter token when it serves as a parameter marker, as explained in "PREPARE" on page 1781.

  Examples:

  ```
  ,       'string'       "fld1"       =       .
  ```

Spaces

A *space* is a sequence of one or more blank characters.

Control characters

A *control character* is a special character that is used for string alignment. Treated similar to a space, a control character does not cause a particular action to occur. The following table shows the control characters that DB2 recognizes and their hexadecimal values.

In an SQL procedure, a *new line control character* is a special character that is used for a new line. The carriage return, new line or next line, and line feed (new line) characters, or the combination of carriage return followed by new line characters, and the combination of carriage return followed by line feed characters, as shown in the following table, are the new line control characters for SQL procedures.

Table 5. Hexadecimal values for the control characters that DB2 recognizes

Control character	EBCDIC hex value	Unicode hex value
Tab	05	09
Form feed	0C	0C
Carriage return	0D	0D
New line or next line	15	C285
Line feed (new line)	25	0A

Tokens, other than string constants and certain delimited identifiers, must not include a control character or space. A control character or space can follow a token. A delimiter token, control character, or a space must follow every ordinary token. If the syntax does not allow a delimiter token to follow an ordinary token, a control character or a space must follow that ordinary token.

Trigraphs

The left bracket ([) and right bracket (]) characters are used in syntax to refer to an array element. Those characters cannot be specified with some CCSIDs. The following trigraphs can be used as an alternative way to specify left and right brackets:

- The string ??(can be specified in place of a left bracket ([).
- The string ??) can be specified in place of a right bracket (]).

Comments

Dynamic SQL statements can include SQL comments. Static SQL statements can include host language comments or SQL comments. Comments can be specified wherever a space can be specified, except within a delimiter token or between the keywords EXEC and SQL. In Java, SQL comments are not allowed within embedded Java expressions. There are two types of SQL comments:

simple comments
Simple comments are introduced with two consecutive hyphens (--). Simple comments cannot continue past the end of the line. For additional information, see "SQL comments" on page 846.

bracketed comments
Bracketed comments are introduced with /* and end with */. A bracketed comment can continue past the end of the line. For additional information, see "SQL comments" on page 846.

Uppercase and lowercase

A token in an SQL statement can include lowercase letters, but lowercase letters in an ordinary token are folded to uppercase. However, lowercase letters are folded to uppercase in a C or Java program only if the appropriate precompiler option is specified. Delimiter tokens are never folded to uppercase.

Example: The following two statements, after folding, are equivalent:

```
select * from DSN8B10.EMP where lastname = 'Smith';
SELECT * FROM DSN8B10.EMP WHERE LASTNAME = 'Smith';
```

Identifiers

An *identifier* is a token used to form a name. An identifier in an SQL statement is an SQL identifier or a host identifier.

See "Limits in DB2 for z/OS" on page 2012 for the identifier length limits that DB2 imposes.

SQL identifiers

SQL identifiers can be *ordinary identifiers* or *delimited identifiers*.

Ordinary identifiers

An *ordinary identifier* is an uppercase letter followed by zero or more characters, each of which is an uppercase letter, a digit, or the underscore character.

An ordinary identifier should not be a reserved word. If a reserved word is used as an identifier in SQL, it must be specified in uppercase and must be a delimited

identifier or specified in a host variable. For a list of reserved words, see “Reserved schema names and reserved words” on page 2019.

SQL ordinary identifiers can contain DBCS characters unless otherwise specified. However, an SQL ordinary identifier cannot contain a mixture of SBCS and DBCS characters.

The following list shows the rules for forming SQL ordinary identifiers:

- The UTF-8 representation of the name must not exceed 128 bytes.
- Continuation to the next line is not allowed.

If the SQL ordinary identifier contains DBCS characters, the following additional rules apply:

- The identifier, if encoded in EBCDIC, must start with a shift-out (X'0E') and end with a shift-in (X'0F'). There must be an even number of bytes between the shift-out and the shift-in. An odd-numbered byte between those shifts must not be a shift-out. DBCS blanks (X'4040' in EBCDIC) are not acceptable between the shift-out and the shift-in.
- The identifiers are not folded to uppercase or changed in any other way.

Example: The following example is an ordinary identifier:

```
SALARY
```

Delimited identifiers

A *delimited identifier* is a sequence of one or more characters enclosed within escape characters.

The escape character is the quotation mark (")[2] except for:

- Dynamic SQL when the field SQL STRING DELIMITER on installation panel DSNTIPF is set to the quotation mark (") and either of these conditions is true:
  - DYNAMICRULES run behavior applies. For a list of the DYNAMICRULES option values that specify run, bind, define, or invoke behavior, see Table 6 on page 75.
  - DYNAMICRULES bind, define, or invoke behavior applies and installation panel field USE FOR DYNAMIC RULES is YES.

  In this case, the escape character is the apostrophe (').

  However, for COBOL application programs, if DYNAMICRULES run behavior does not apply and installation panel field USE FOR DYNAMICRULES is NO, a COBOL compiler option specifies whether the escape character is the quotation mark or apostrophe.
- Static SQL in COBOL application programs. A COBOL compiler option specifies whether the escape character is the quotation mark (") or the apostrophe (').

A delimited identifier can be used when the sequence of characters does not qualify as an ordinary identifier. Such a sequence, for example, could be an SQL reserved word, or it could begin with a digit. Two consecutive escape characters are used to represent one escape character within the delimited identifier. A delimited identifier that contains EBCDIC DBCS characters also must contain the necessary shift characters.

2. In CCSID 1026 and CCSID 1155, the code point for the quotation mark can be X'7F' or X'FC'. However, if the beginning delimiter is X'7F', the ending delimiter must also be X'7F'. If the beginning delimiter is X'FC', ending delimiter must also be X'FC'.

Leading and embedded blanks in the sequence are significant. Trailing blanks in the sequence are not significant. The length of a delimited identifier does not include the starting and ending escape characters. Embedded escape characters (that appear as two characters) are counted in the length as a single character.

Example: If the escape character is the quotation mark, the following example is a delimited identifier:

```
"VIEW"
```

Host identifiers

A *host identifier* is a name declared in the host program.

The rules for forming a host identifier are the rules of the host language. In non-Java programs, do not use names beginning with 'DB2', 'SQ'[3], 'SQL', 'sql', 'RDI', or 'DSN' because precompilers generate host variable names that begin with these characters. In Java, do not use names beginning with '__sJT_'.

Restrictions for distributed access

To use certain identifiers in distributed access, those identifiers need to be restricted to certain characters.

DB2's internal processing of distributed access must sometimes convert the identifiers for *authorization-name*, *procedure-name*, and *schema-name* between CCSIDs. If there is any possibility that these identifiers will be used in distributed access, restrict the identifiers to characters whose representation in Unicode UTF-8 have code points in the range 0 through 127. You do not need to enter the identifiers in Unicode; this restriction refers to conversion that DB2 performs internally.

Naming conventions

The rules for forming a name depend on the type of the object designated by the name.

The syntax diagrams use different terms for different types of names. The following list defines these terms.

alias-name

A qualified or unqualified name that designates an alias. A fully qualified alias name is a three-part name. The first part is a location name that designates the DBMS at which the sequence is defined. The second part is a schema name. The third part is an SQL identifier. A period must separate each of the parts.

A two-part sequence is implicitly qualified by the location name of the current server. The first part is a schema name. The second part is an SQL identifier. A period must separate the two parts.

A one-part or unqualified sequence name is an SQL identifier with two implicit qualifiers. The first implicit qualifier is the location name of the current server. The second is a schema name, which is determined by the rules specified in “Unqualified alias, index, JAR file, sequence, table, trigger, and view names” on page 66.

See “Aliases” on page 67 for additional information about aliases.

3. 'SQ' is allowed in C, COBOL, and REXX.

array-type-name
A qualified or unqualified name that designates an array type.

A qualified array type name is a two-part name. The first part is the schema name of the array type. The second part is an SQL identifier. A period must separate each of the parts.

An unqualified array type name is an SQL identifier with an implicit qualifier. The implicit qualifier is the schema name, which is determined by the context in which the array type appears, as described by the rules in "Qualification of unqualified object names" on page 65.

authorization-name
An SQL identifier that designates a set of privileges. It can also designate a user, a group of users, or a role. For a user or a group of users, DB2 does not control this property. For a role, DB2 does control this property. See "Authorization IDs, roles, and authorization names" on page 72 for the distinction between an authorization name and an authorization ID.

aux-table-name
A qualified or unqualified name that designates an auxiliary table. The rules for the name are the same as the rules for *table-name*. See table-name.

bpname
A name that identifies a buffer pool. The following list shows the names of the different buffer pool sizes.
4KB BP0, BP1, BP2, ..., BP49
8KB BP8K0, BP8K1, BP8K2, ..., BP8K9
16KB BP16K0, BP16K1, BP16K2, ..., BP16K9
32KB BP32K, BP32K1, BP32K2, ..., BP32K9

built-in-type
A qualified or unqualified name that identifies an IBM-supplied data type. A qualified name is SYSIBM followed by a period and the name of the built-in data type. An unqualified name has an implicit qualifier, the schema name, which is determined by the rules in "Qualification of unqualified object names" on page 65.

catalog-name
An SQL identifier that designates an integrated catalog facility catalog. The identifier must start with a letter and must not include special characters.

clone-table-name
A qualified or unqualified name that designates the name of a clone table. See the definition of *table-name* for more information about qualification of table names.

collection-id
An SQL identifier that identifies a collection of packages, such as a collection ID as a qualifier for a package ID. Refer to *DB2 Command Reference* for naming conventions.

column-name
A qualified or unqualified name that designates a column of a table or view.

A qualified column name is a qualifier followed by a period and an SQL identifier. The qualifier is a table name, a view name, a synonym, an alias, or a correlation name. The unqualified column name is an SQL identifier.

constraint-name
An SQL identifier that designates a primary key, check, referential, or unique constraint on a table.

correlation-name
An SQL identifier that designates a table, a view, or individual rows of a table or view.

context-name
An unqualified SQL identifier that designates a trusted context.

cursor-name
An SQL identifier that designates an SQL cursor. In SQLJ, *cursor-name* is a host variable (with no indicator variable) that identifies an instance of an iterator.

database-name
An SQL identifier that designates a database. The identifier must start with a letter and must not include special characters.

descriptor-name
A host identifier that designates an SQL descriptor area (SQLDA). See "References to host variables" on page 215 for a description of a host identifier. A descriptor name never includes an indicator variable.

distinct-type-name
A qualified or unqualified name that designates a distinct type.

A qualified distinct type name is a two-part name. The first part is the schema name of the distinct type. The second part is an SQL identifier. A period must separate each of the parts.

An unqualified distinct type name is an SQL identifier with an implicit qualifier. The implicit qualifier is the schema name, which is determined by the context in which the distinct type appears as described by the rules in "Unqualified type, function, procedure, global variable, and specific names" on page 66.

external-program-name
A name that specifies the program that runs when the function is invoked or the procedure name is specified in a CALL statement.

function-name
A qualified or unqualified name that designates a user-defined function, a cast function that was generated when a distinct type was created, or a built-in function.

A qualified function name is a two-part name. The first part is the schema name of the function. The second part is an SQL identifier. A period must separate each of the parts.

An unqualified function name is an SQL identifier with an implicit qualifier. The implicit qualifier is the schema name, which is determined by the context in which the unqualified name appears as described by the rules in "Unqualified type, function, procedure, global variable, and specific names" on page 66.

host-label
A token that designates a label in a host program.

host-variable
A sequence of tokens that designates a host variable. A host variable includes at least one host identifier, as explained in "References to host variables" on page 215.

index-name
A qualified or unqualified name that designates an index.

A qualified index name is an authorization ID or schema name followed by a period and an SQL identifier.

An unqualified index name is an SQL identifier with an implicit qualifier. The implicit qualifier is an authorization ID, which is determined by the context in which the unqualified name appears as described by the rules in "Qualification of unqualified object names" on page 65.

For an index on a declared temporary table, the qualifier must be SESSION.

location-name
An SQL identifier that designates the name of a location. A location name is 1 to 16 bytes, does not include alphabetic extenders, lowercase letters, or Katakana characters. The characters allowed in the delimited form are the same as those allowed in the ordinary form.

mask-name
A qualified or unqualified name that designates a mask.

A qualified mask name is a two-part name. The first part is the schema name. The second part is an SQL identifier. A period must separate each of the parts.

A one-part or unqualified mask name is an SQL identifier with an implicit qualifier. The implicit qualifier is an authorization ID, which is determined by the context in which the unqualified name appears as described by the rules in "Qualification of unqualified object names" on page 65.

package-name
A qualified or unqualified name that designates a package. The unqualified form of a *package-name* is an SQL identifier. A *package-name* must not be a delimited identifier that includes lowercase letters or special characters. A *package-name* in an SQL statement must be qualified. In some contexts outside of SQL, a package name can be specified as an unqualified name.

parameter-name
An SQL identifier that designates a parameter in an SQL procedure or SQL function.

permission-name
A qualified or unqualified name that designates a permission.

A qualified permission name is a two-part name. The first part is the schema name. The second part is an SQL identifier. A period must separate each of the parts.

A one-part or unqualified permission name is an SQL identifier with an implicit qualifier. The implicit qualifier is an authorization ID, which is determined by the context in which the unqualified name appears as described by the rules in "Qualification of unqualified object names" on page 65.

plan-name
An SQL identifier that designates an application plan. The identifier must not be a delimited identifier that includes lowercase letters or special characters.

procedure-name
A qualified or unqualified name that designates a stored procedure.

A fully qualified procedure name is a three-part name. The first part is a location name that identifies the DBMS at which the procedure is stored. The second part is the schema name of the stored procedure. The third part is an SQL identifier. A period must separate each of the parts in a qualified name.

A two-part procedure name is implicitly qualified with the location name of the current server. The first part is the schema name of the stored procedure. The second part is an SQL identifier. A period must separate the two parts.

A one part, or unqualified, procedure name is an SQL identifier with two implicit qualifiers. The first implicit qualifier is the location name of the current server. The second implicit qualifier is the schema name, which is determined by the context in which the unqualified name appears, as described by the rules in "Qualification of unqualified object names" on page 65.

The SQL identifier in a qualified or unqualified name must not be an asterisk (*).

profile-name
An SQL identifier that corresponds to a RACF profile name.

program-name
An SQL identifier that designates an exit routine.

role-name
An unqualified SQL identifier that designates a role. The identifier cannot begin with the characters SYS and cannot be ACCESSCTRL, DATAACCESS, DBADM, DBCTRL, DBMAINT, NONE, NULL, PACKADM, PUBLIC, SECADM, or SQLADM.

routine-version-id
An SQL identifier of up to 64 EBCDIC bytes that designates a version of a routine. The UTF-8 representation of the name must not exceed 122 bytes.

savepoint-name
An unqualified SQL identifier that designates a savepoint.

schema-name
An SQL identifier that provides a logical grouping for SQL objects. A *schema-name* is used as a qualifier of the name of SQL objects.

seclabel-name
A string that corresponds to the value of the RACF security label. It is recommended that name not include national characters (@ (X'7C'), # (X'7B'), or $ (X'5B')). If the table is a Unicode table and the security label name does include national characters, an error might be issued if substitution occurs when DB2 converts the value from EBCDIC to Unicode.

sequence-name
A qualified or unqualified name that designates a sequence.

A qualified sequence name is a two-part name. The first part is the schema name. The second part is an SQL identifier. A period must separate each of the parts.

A one-part or unqualified sequence name is an SQL identifier with an implicit qualifier. The implicit qualifier is an authorization ID, which is determined by the context in which the unqualified name appears as described by the rules in "Unqualified alias, index, JAR file, sequence, table, trigger, and view names" on page 66.

server-name
An SQL identifier that designates an application server. The identifier must start with a letter and must not include lowercase letters or special characters.

session-variable-name
A qualified or unqualified name that designates a global variable.

A qualified global variable name is a two-part name. The first part is the schema name of the global variable. The second part is an SQL identifier. A period must separate each of the parts.

An unqualified global variable name is an SQL identifier with an implicit qualifier. The implicit qualifier is the schema name, which is determined by the context in which the unqualified name appears as described by the rules in "Unqualified type, function, procedure, global variable, and specific names" on page 66.

specific-name
A qualified or unqualified name that designates a unique name for a user-defined function.

A qualified specific name is a two-part name. The first part is the schema name. The second part is an SQL identifier, and it must not be an asterisk (*). A period must separate each of the parts.

An unqualified specific name is an SQL identifier with an implicit qualifier. The implicit qualifier is the schema name, which is determined by the context in which the unqualified name appears as described by the rules in "Unqualified type, function, procedure, global variable, and specific names" on page 66.

A specific name can be used to identify a function to alter, comment on, drop, grant privileges on, revoke privileges from, or be the source function for another function. A specific name cannot be used to invoke a function. In addition to being used in certain SQL statements, a specific name must be used in DB2 commands to uniquely identify a function.

SQL-condition-name
An SQL identifier that designates a condition in an SQL function or an SQL procedure.

SQL-label
An SQL identifier that designates a label in an SQL function or an SQL procedure.

SQL-parameter-name
A qualified or unqualified name that designates a parameter in the SQL routine body of an SQL function or SQL procedure. The unqualified form of an *SQL-parameter-name* is an SQL identifier. The qualified form is a *function-name* or *procedure-name* followed by a period and an SQL identifier.

SQL-variable-name
A qualified or unqualified name that designates a variable in an SQL routine body. The unqualified form of an *SQL-variable-name* is an SQL identifier. The qualified form is an *SQL-label* followed by a period (.) and an SQL identifier.

statement-name
An SQL identifier that designates a prepared SQL statement.

stogroup-name
An SQL identifier that designates a storage group.

synonym
An SQL identifier that designates a synonym, a table, or a view. The table or view must exist at the current server. A qualified name is never interpreted as a synonym.

table-name
A qualified or unqualified name that designates a table.

A fully qualified table name is a three-part name. The first part is a location name that designates the DBMS at which the table is stored. The second part is a schema name. The third part is an SQL identifier. A period must separate each of the parts.

A two-part table name is implicitly qualified by the location name of the current server. The first part is a schema name. The second part is an SQL identifier. A period must separate the two parts.

A one-part or unqualified table name is an SQL identifier with two implicit qualifiers. The first implicit qualifier is the location name of the current server. The second is a schema name, which is determined by the rules set forth in "Unqualified alias, index, JAR file, sequence, table, trigger, and view names" on page 66. For a declared temporary table, the qualifier (the second part in a three-part name and the first part in a two-part name) must be SESSION. For complete details on specifying a name when a declared temporary table is defined and then later referring to that declared temporary table in other SQL statements, see "DECLARE GLOBAL TEMPORARY TABLE" on page 1547.

table-space-name
An SQL identifier that designates a table space of an identified database. The identifier must start with a letter and must not include special characters. If a database is not identified, DSNDB04 is implicit.

trigger-name
A qualified or unqualified name that designates a trigger.

A qualified trigger name is a two-part name. The first part is the schema name of the trigger. The second part is an SQL identifier. A period must separate each of the parts.

An unqualified trigger name is an SQL identifier with an implicit qualifier. The implicit qualifier is the schema name, which is determined by the context in which the unqualified name appears as described by the rules in "Unqualified alias, index, JAR file, sequence, table, trigger, and view names" on page 66.

view-name
A qualified or unqualified name that designates a view.

A fully qualified view name is a three-part name. The first part is a location name that designates the DBMS where the view is defined. The second part is a schema name. The third part is an SQL identifier. A period must separate each of the parts.

A two-part view name is implicitly qualified by the location name of the current server. The first part is a schema name. The second part is an SQL identifier. A period must separate the two parts.

A one-part or unqualified view name is an SQL identifier with two implicit qualifiers. The first implicit qualifier is the location name of the current server. The second is a schema name, which is determined by the context in which the unqualified name appears as described by the rules in "Unqualified alias, index, JAR file, sequence, table, trigger, and view names" on page 66.

XML-attribute-name
An identifier that is used as an XML attribute name.

XML-element-name
An identifier that is used as an XML element name.

SQL path

The *SQL path* is an ordered list of schema names. DB2 uses the path to resolve the schema name for certain unqualified object names that appear in any context other than as the main object of an ALTER, CREATE, DROP, COMMENT, GRANT, RENAME, or REVOKE statement.

DB2 uses the path to resolve the schema name for the following object names:

- data types (both built-in types and distinct types)
- functions
- stored procedures
- global variables

Searching through the path from left to right, DB2 implicitly qualifies the object name with the first schema name in the SQL path that contains the same object with the same unqualified name for which the user has appropriate authorization. For functions, DB2 uses a process called function resolution in conjunction with the SQL path to determine which function to choose because several functions with the same name and number of parameters but different parameter data types might be defined in the same schema or other schemas in the SQL path. (For details, see "Function resolution" on page 234.) For procedures, DB2 selects a matching procedure name only if the number of parameters is also the same.

The SQL path does not apply to unqualified procedure names in ASSOCIATE LOCATOR and DESCRIBE PROCEDURE statements. For these statements, an implicit schema name is not generated.

For an example of how DB2 uses the SQL path to resolve the schema name, assume that the SQL path is SMITH, XGRAPHIC, SYSIBM, and that an unqualified distinct type name MYTYPE was specified. DB2 looks for MYTYPE first in schema SMITH, then XGRAPHIC, and then SYSIBM.

The PATH option establishes the SQL path that is used to resolve:

- Unqualified data type, global variable, and function names in static SQL statements

- Unqualified procedure names in SQL CALL statements that specify the procedure name as an identifier token (CALL *procedure-name*)

If the PATH option was not specified when the plan or package was created or last rebound or when native SQL procedure was defined or last changed, the default value of the SQL path is: SYSIBM, SYSFUN, SYSPROC, *plan or package qualifier*.

The CURRENT PATH special register determines the SQL path used to resolve:

- Unqualified data type, global variable, and function names in dynamic SQL statements
- Unqualified procedure names in SQL CALL statements that specify the procedure name in a host variable (CALL *host-variable*)

Generally, the initial value of the CURRENT PATH special register is one of the following:

- The value of the PATH option
- "SYSIBM", "SYSFUN", "SYSPROC", "SYSIBMADM", *value of CURRENT SQLID special register* if the PATH option was not specified.

If schema "SYSIBM", "SYSFUN", "SYSPROC", "SYSIBMADM" is not explicitly specified in the SQL path, the schema is implicitly assumed at the front of the path; if all are not specified, they are assumed in the order of "SYSIBM", "SYSFUN", "SYSPROC", "SYSIBMADM".

For example, assume that the SQL path is explicitly specified as SYSIBM, GEORGIA, SMITH. As implicitly assumed schemas, SYSFUN, SYSPROC, and SYSIBMADM are added to the beginning of the explicit path effectively making the path:

SYSFUN, SYSPROC, SYSIBMADM, SYSIBM, GEORGIA, SMITH

For more information about the SQL path for dynamic SQL, see “CURRENT PATH” on page 184 and “SET PATH” on page 1921.

Resolution of unqualified object names

Most object names are implicitly or explicitly qualified with a schema name. Synonyms are an exception.

A synonym has a single part name. When DB2 encounters an unqualified name, DB2 must determine which object to process. This process is called *name resolution*.

When DB2 encounters a single part name in a context where an alias, table, view, or synonym can be specified, DB2 first checks to see if the name refers to a synonym that is defined by the current user.

Qualification of unqualified object names

Unqualified object names, other than synonyms, are implicitly qualified. The rules for qualifying a name differ depending on the type of object that the name identifies.

Unqualified alias, index, JAR file, sequence, table, trigger, and view names

Unqualified alias, index, JAR file, sequence, table, trigger, and view names are implicitly qualified by the default schema.

The default schema is determined as follows:

- For static SQL statements, the default schema is the identifier specified in the QUALIFIER option of the BIND subcommand or the CREATE PROCEDURE or ALTER PROCEDURE statement (for a native SQL procedure). If this option is not in effect for the plan, package, or native SQL procedure, the default schema is the authorization ID of the owner of the plan, package, or native SQL procedure.
- For dynamic SQL statements, the behavior as specified by the combination of the DYNAMICRULES option and the run time environment determines the default schema. (For a list of these behaviors and the DYNAMICRULES values that determine them, see Table 6 on page 75).
  - If DYNAMICRULES *run behavior* applies, the default schema is the schema in the CURRENT SCHEMA special register. Run behavior is the default.
  - If *bind behavior* applies, the default schema is the identifier that is implicitly or explicitly specified in the QUALIFIER option, as explained for static SQL statements.
  - If *define behavior* applies, the default schema is the owner of the function or stored procedure (the owner is the definer).
  - If *invoke behavior* applies, the default schema is the authorization ID of the invoker of the function or stored procedure.

  Exception: For bind, define, and invoke behavior, the default schema of PLAN_TABLE, DSN_STATEMNT_TABLE, and DSN_FUNCTION_TABLE (output from the EXPLAIN statement) is always the value in special register CURRENT SQLID.

Related reference:

QUALIFIER bind option (DB2 Commands)

DYNAMICRULES bind option (DB2 Commands)

"CREATE PROCEDURE (SQL - native)" on page 1350

"ALTER PROCEDURE (SQL - native)" on page 947

"CURRENT SQLID" on page 193

"EXPLAIN" on page 1642

Unqualified type, function, procedure, global variable, and specific names

The qualification of unqualified type (built-in type, distinct type, or array type), function, stored procedure, global variable, and specific names depends on the SQL statement in which the unqualified name appears.

- If an unqualified name is the main object of an ALTER, CREATE, COMMENT, DROP, GRANT, or REVOKE statement, the name is implicitly qualified with a schema name as follows:
  - In a static statement, the implicit schema name is the identifier specified in the QUALIFIER option of the BIND subcommand or the CREATE PROCEDURE or ALTER PROCEDURE statement (for a native SQL

procedure). If this option is not in effect for the package or procedure, the implicit qualifier is the authorization ID of the owner of the package or procedure.
  - In a dynamic statement, the implicit schema name is the schema in the CURRENT SCHEMA special register.
- Otherwise, the implicit schema name for the unqualified name is determined as follows:
  - For distinct type and array type names, DB2 searches the SQL path and selects the first schema in the path such that the data type exists in the schema and the user has authorization to use the type.
  - For global variable names, DB2 searches the SQL path and selects the first schema in the path such that the global variable exists in the schema and the user has authorization to use the global variable.
  - For function names, DB2 uses the SQL path in conjunction with function resolution, as described in "Function resolution" on page 234.
  - For stored procedure names in CALL statements, DB2 searches the SQL path and selects the first schema in the path such that the schema contains a procedure with the same name and number of parameters and the user has authorization to use the procedure.
  - For stored procedure names in ASSOCIATE LOCATORS and DESCRIBE PROCEDURE statements, DB2 does not use the SQL path because an implicit schema name is not generated for these statements.

  For information about the SQL path, see "SQL path" on page 64.

Aliases

An *alias* is an alternative name for an object such as a table, view, sequence, or another alias. It can be used to reference an object wherever that object can be referenced directly.

The option of referencing an object by an alias is not explicitly shown in the syntax diagrams or mentioned in the description of SQL statements.

Like tables, views, and sequences, an alias can be created, dropped, and associated with a comment. No authority is necessary to use an alias. However, access to the objects that are referred to by the alias still requires the appropriate authorization for the current statement.

An alias is created using the CREATE ALIAS statement.

An alias name designates an alias when it is preceded by the keyword ALIAS, as in CREATE ALIAS, DROP ALIAS, COMMENT ON ALIAS, and LABEL for an ALIAS. In all other contexts, an alias name designates a table, a view, or a sequence. For example, `COMMENT ON ALIAS A` specifies a comment about the alias A, whereas `COMMENT ON TABLE A` specifies a comment about the table or view designated by A.

An alias for a table or a view can be defined at a local server to refer to a table or a view that is at the current server or a remote server. An alias name for a table or view can be used wherever the table name or view name can be used to refer to the table or view in an SQL statement. The rules for forming an alias name for a table or view are the same as the rules for forming a table name or a view name. A

fully qualified alias name (a three-part name) can refer to an alias at a remote server. However, the table or view identified by the alias at the remote server must exist at the remote server.

An alias for a sequence can be defined at the current server. An alias name for a sequence can be used wherever the sequence name can be used to refer to the sequence in an SQL statement. The rules for forming an alias name for a sequence are the same as the rules for forming a sequence name.

Statements that use three-part names and refer to distributed data result in DRDA access to the remote site. DRDA access for three-part names is used when the package that contains the query to distributed data is bound using the bind option DBPROTOCOL(DRDA), or the value of the DATABASE PROTOCOL field on installation panel DSNTIP5 is DRDA. When an application program uses three-part name aliases for remote table or view objects and DRDA access, the application program must be bound at each location that is specified in the three-part name. Also, each alias must be defined at the local site. An alias at a remote site can refer to another server if a referenced alias eventually refers to a table or view.

The effect of using an alias in an SQL statement is the same as text substitution. For example, if A is an alias for table Q.T, one of the steps involved in the preparation of `SELECT * FROM A` is the replacement of 'A' by 'Q.T'.

If an alias is defined as a public alias, it can be referenced by its unqualified name without any impact from the current default schema name. It can also be referenced using the schema qualifier SYSPUBLIC.

Related concepts:

"Synonyms"

Related reference:

"CREATE ALIAS" on page 1154

Synonyms

A synonym is an alternate name for a table or view. A synonym can be used to reference a table or view in cases where an existing table or view can be referenced.

Important: Synonyms are similar to aliases, but are only supported for compatibility with previous releases. Synonyms behave differently with DB2 for z/OS than with the other DB2 family products. Do not use synonyms when writing new SQL statements or when creating portable applications. Use aliases instead.

The option of referencing a table or view by an synonym is not explicitly shown in the syntax diagrams or mentioned in the description of SQL statements. But synonyms can be referred to in an SQL statement, with one exception: a synonym cannot be used in the CREATE SYNONYM statement.

Like tables and views, a synonym can be created, dropped, and associated with a comment. No authority is necessary to use a synonym. However, access to the tables and views that are referenced by the synonym still requires the appropriate authorization for the current statement.

A synonym is created with the CREATE SYNONYM statement.

A synonym name designates a synonym when it is preceded by the keyword SYNONYM, as in CREATE SYNONYM and DROP SYNONYM. In all other contexts, a synonym designates a table or a view. In all other contexts, a synonym designates a local table or view and can be used wherever the name of a table or view can be used in an SQL statement.

The effect of using a synonym in an SQL statement is the same as text substitution. For example, if S is a synonym for Q.T, one of the steps involved in the preparation of `SELECT * FROM S` is the replacement of 'S' by 'Q.T'.

The differences between aliases and synonyms are as follows:

- Authorization or the CREATE ALIAS privilege is required to define an alias. No authorization is required to define a synonym.
- An alias can be defined on the name of a table or view, including tables and views that are not at the current server. A synonym can only be defined on the name of a table or view at the current server.
- An alias can be defined on an undefined name. A synonym can only be defined on the name of an existing table or view.
- Dropping a table or view has no effect on its aliases. But dropping a table or view does drop its synonyms.
- An alias is a qualified name that can be used by any authorization ID. A new alias cannot have the same fully qualified name as an existing alias, table, or view, and a new unqualified alias name cannot have the same name as an existing synonym.
- A synonym is an unqualified name that can only be used by the authorization ID that created it. A new synonym cannot have the same name as an existing synonym, or the unqualified name of an existing alias, table, or view.
- An alias defined at one DB2 subsystem can be used at another DB2 subsystem. A synonym can only be used at the DB2 subsystem where it is defined.
- When an alias is used, an error occurs if the name that it designates is undefined or is the name of an alias at the current server. (The alias can represent another alias at a different server, which can represent yet another alias at yet another server as long as eventually a referenced alias represents a table or view.) When a synonym is used, this error cannot occur.
- A synonym cannot be created in a trusted context that has ROLE AS OBJECT OWNER in effect.
- An alias specified in the CREATE SYNONYM statement must identify a table or view at the current server. The synonym is defined on the name of that table or view.
- A synonym specified in the CREATE ALIAS statement defines an alias on the name of the table or view identified by the synonym.

Related concepts:

"Aliases" on page 67

Related reference:

"CREATE SYNONYM" on page 1386

Authorization, privileges, permissions, masks, and object ownership

Users (as identified by an authorization ID) can successfully execute SQL statements only if they have the authority to perform the specified operation. For example, to create a table, a user must be authorized to create tables.

The two forms of authorization are administrative authority and privileges.

Administrative authority

The holder of administrative authority is charged with the task of controlling DB2 and is responsible for the safety and integrity of the data.

Those with SYSADM authority implicitly have all privileges on all objects and control who will have access to DB2 and the extent of this access.

Those with SECADM authority manage security policies by enforcing row and column access control for tables that contain sensitive data. They define row permissions and column masks, which describe how tables that use row or column access controls should be accessed and which determine whether a trigger or a user-defined function is considered secure for those tables.

Privileges

Privileges are those activities that a user is allowed to perform. Authorized users can create objects, have access to objects that they own, and can pass on privileges on the objects that they own to other users by using the GRANT statement. Privileges can be granted to specific users or to PUBLIC. PUBLIC specifies that a privilege is granted to all users (including to future users).

The REVOKE statement can be used to revoke previously granted privileges.

Row permissions and column masks

A *row permission* is a database object that expresses an access control rule for a row of a specific table. A row permission is in the form of a search condition that describes to which rows users have access. Row permissions are applied after table privileges (like SELECT or INSERT) are checked.

A *column mask* is a database object that expresses an access control rule for a specific column in a table. A column mask is in the form of a CASE expression that describes to which column values users have access. Column masks are applied after table privileges (like SELECT or INSERT) are checked.

Row permissions and column masks can be created, changed, and dropped only by those with SECADM authority by using the CREATE MASK, CREATE PERMISSION, and DROP statements. The definition of a permission or a mask can reference other objects. Those with SECADM authority do not need additional privileges to reference those objects, such as SELECT privilege to retrieve from a table or EXECUTE privilege to invoke a user-defined function, in the definition of the row permission or column mask. Multiple row permissions and column masks can be created for a table. Only one column mask can be created for each column in a

table. A row permission or a column mask can be created before row or column access control is enforced for a table. The definition of the row permission and the column mask is stored in the DB2 catalog. However, the permission and the mask do not take effect until the ALTER TABLE statement with the ACTIVATE ROW ACCESS CONTROL clause is used to enforce row access control or the ACTIVATE COLUMN ACCESS CONTROL clause is used to enforce column access control on the table.

When an ALTER TABLE statement is used to explicitly activate row access control for a table, a default row permission is implicitly created for the table which prevents all access to the table. After row access controls have been activated for a table, if the table is referenced explicitly in a data change statement and if multiple row permissions are defined for the table, a row access control search condition is derived by using the logical OR operator with the search condition of each defined row permission.

When an ALTER TABLE statement is used to explicitly activate column access control for a table, access to the table is not restricted. However, if the table is referenced in a data change statement, all column masks that have been created for the table are applied to mask the column values that are referenced in the output of the queries or to determine the column values that are used in the data change statements.

The authorization ID or role for the statement does not need authority to reference objects that are specified in the definition of the row permission or column mask.

Object ownership

When an object is created, one authorization ID is assigned ownership of the object. Ownership means that the user is authorized to reference the object in any applicable SQL statement. The privileges on the object can be granted by the owner, and cannot be revoked from the owner. Owners of views only receive the level of privileges that they have on the underlying table or view. The owner of the object that is being created is determined as follows:

- If the schema qualifier is not explicitly specified, the owner depends on how the CREATE statement is issued:
  - If the CREATE statement is embedded in a program, the owner of the object that is being created is the authorization ID that serves as the implicit qualifier for unqualified object names. This is the authorization ID that is in the QUALIFIER option when the plan, package, or native SQL procedure (that contains the CREATE statement) is created or last changed. If the QUALIFIER option is not used, the owner of the object is the authorization ID in the OWNER option when the plan, package, or native SQL procedure is created or last changed. If the OWNER option is not used, the owner is the owner of the plan, package, or native SQL procedure. If the plan or package was last bound in a trusted context that is defined with the ROLE AS OBJECT OWNER clause, a role is the owner.
  - If the CREATE statement is dynamically prepared, the owner of the object that is being created is the authorization ID of the process.
  - If the CREATE statement is execute in a trusted context that is defined with the ROLE AS OBJECT OWNER clause, the role of the primary authorization ID is the owner.
- If the schema qualifier is explicitly specified, the owner depends on the type of object that is being created unless the CREATE statement is executed in a trusted context that is defined with the ROLE AS OBJECT

OWNER clause. When the CREATE statement is executed in a trusted context that is defined with the ROLE AS OBJECT OWNER clause, the owner of the object is determined as follows:

- If the CREATE statement is embedded in a program, the role that owns the plan or package is the owner of the object.
- If the CREATE statement is dynamically prepared, the primary authorization ID is the owner.

If the schema qualifier is explicitly specified, and the CREATE statement is not executed in a trusted context that is defined with the ROLE AS OBJECT OWNER clause, the owner depends on the type of object that is being created: :

- For an alias, auxiliary table, created global temporary table, table, or view, the owner of the object that is being created is the same as the explicit schema name.
- For a user-defined distinct type, user-defined function, procedure, sequence, JAR files, or trigger, the owner of the object that is being created is the authorization ID of the process.

The rules that determine ownership of row permissions and column masks are the same as those that determine ownership of objects like user-defined distinct types, user-defined functions, procedures, sequences, JAR files, or trigger.

The owner of a row permission or a column mask does not have implicit owner privileges. Only users with SECADM authority can manage and maintain row permissions and column masks.

Authorization IDs, roles, and authorization names

Processes can successfully execute SQL statements only if they have the necessary authority. A process derives this authority from its authorization IDs. An authorization ID can also designate a user, a group of users, or a role.

An *authorization ID* is a character string that is associated with a process that is checked to determine the authority to perform a specified operation.

DB2 does not control the association of users to user groups. However, DB2 does control the association between users and roles when a trusted context is defined.

DB2 uses authorization IDs to provide authorization checking of SQL statements.

Whenever a connection is established between DB2 and a process, DB2 obtains an authorization ID and passes it to the authorization connection or sign-on exit routine. The list of one or more authorization IDs that is returned by the exit routine are used as the authorization IDs of the process. If the process is running in a trusted context with a role, the authorization IDs of the process includes this role.

Every process has exactly one primary authorization ID. Any other authorization IDs of a process are secondary authorization IDs. The use of these authorization IDs depends on the type of process (bind process, application process, or process involved in the creation of objects).

Primary authorization ID
An authorization ID that is used to established a connection between DB2 and an application process.

Secondary authorization ID
An authorization ID that is associated with a primary authorization ID.

Secondary authorization IDs includes all the authorization IDs that have been associated with a primary authorization ID by the connection or sign-on authorization exit routine, the CURRENT SQLID (when different from the primary authorization ID), and other authorization IDs like the stored procedure definer and call package owner for stored procedure package checking.

Authorization ID of the process
The user's primary and secondary authorization IDs. If the process is running in a trusted context with a role, the authorization IDs of the process includes this role.

A *role* is a database entity that groups together one or more privileges. A role is available only when the process is run in a trusted context. Users are associated with a role in the definition of a trusted context.

A trusted context can have a default role, specific roles for individual users, or no roles at all. A user in a trusted context can have only one active role. This is the role that is specifically defined for the user or the default role of the trusted context. The following restrictions apply to roles:

- A role cannot be a primary authorization ID.
- A role cannot be set by using a SET CURRENT SQLID statement.
- A role can be the schema qualifier of an object. However, when it is used as a schema qualifier, a role is considered to be a character string and does not add any implicit schema privileges (ALTERIN, CREATEIN, or DROPIN) to this role.
- A role must already exist for privileges to be granted to it.

The role that is in effect for a user is considered to be one of the secondary authorization IDs of the user.

Do not confuse an *authorization-name* that is specified in an SQL statement with an authorization ID of a process.

Example: Assume that SMITH is your TSO logon, DYNAMICRULES run behavior is in effect, and you execute the following statements interactively:

```
CREATE TABLE TDEPT LIKE DSN8B10.DEPT;
GRANT SELECT ON TDEPT TO KEENE;
```

Also assume that your site has not replaced the default exit routine for connection authorization and that you have not executed the SET CURRENT SQLID statement. Thus, when the GRANT statement is prepared and executed by SPUFI, the SQL authorization ID is SMITH. KEENE is an authorization name that is specified in the GRANT statement.

Authorization to execute the GRANT statement is checked against SMITH. The authorization rule is that the privilege set that is designated by SMITH must include the SELECT privilege with the GRANT option on SMITH.TDEPT. No check that involves KEENE is performed. If the GRANT statement specifies a role, the existence of the role is checked.

Authorization IDs and schema names

An authorization ID that has the same name as the name of a schema implicitly has certain privileges for that schema.

If an authorization ID is not a role and has the same name as the name of a schema, that authorization ID implicitly has the following privileges for that schema:

- CREATEIN privilege
- ALTERIN privilege
- DROPIN privilege

Authorization IDs and statement preparation

The authorization ID that is specified as the owner of the plan or package must be one of the authorization IDs of the bind process. The owner can be set to any value if one of the authorization IDs of the bind process has SYSADM or SYSCTRL authority.

A process that creates a plan or package is called a *bind process*. The connection with DB2 is the result of the execution of a BIND or REBIND subcommand. Both subcommands allow for the specification of the authorization ID of the owner of the plan or package.

BINDAGENT can specify an owner other than himself (or one of his representatives), but it has to be someone that granted him BINDAGENT. The default owner for BIND is the primary authorization ID. The default owner for REBIND is the previous owner of the plan or package (ownership is unchanged if an owner is not explicitly specified). If the BIND or REBIND is performed in a trusted context that is defined with the ROLE AS OBJECT OWNER clause, the owner of the plan or package is a role. If the OWNER bind option is specified, the role that is specified in it is the owner, otherwise the role that performs the bind or rebind becomes the owner.

The authorization ID that is used for the authorization checking of embedded SQL statements is that of the owner of the plan or package. If the application is bound in a trusted context using the ROLE AS OBJECT OWNER clause, the authorization ID that is used for authorization checking is the role that owns the plan or package, otherwise the authorization ID is the authorization ID of the owner of the plan or package. If an embedded SQL statement refers to tables or views at a DB2 subsystem other than the one at which the plan or package is bound, the authorization checking is deferred until run time. For more information on this, see "Authorization IDs and remote execution" on page 77.

If VALIDATE(BIND) is specified, the privileges required to use or manipulate objects at the DB2 subsystem at which the plan or package is bound must exist at bind time. If the privileges or the referenced objects do not exist and SQLERROR(NOPACKAGE) is in effect, the bind operation is unsuccessful. If SQLERROR(CONTINUE) is specified, then the bind is successful and any statements in error are flagged. If any statements in error are flagged, an error will occur when you attempt to execute them at run time.

If a plan or package is bound with VALIDATE(RUN), authorization checking is still performed at bind time, but the referenced objects and the privileges required to use these objects need not exist at this time. If any privilege required for a statement does not exist at bind time, an authorization check is performed

whenever the statement is first executed within a unit of work, and all privileges required for the statement must exist at that time. If any privilege does not exist, execution of the statement is unsuccessful. When the authorization check is performed at run time, it is performed against the plan or package owner, not the SQL authorization ID. For the effect of this option on cursors, see "DECLARE CURSOR" on page 1535.

Related reference:

The DSN command and its subcommands (DB2 Commands)

Authorization IDs and dynamic SQL

The bind option DYNAMICRULES determines the authorization ID that is used for checking authorization when dynamic SQL statements are processed. The set of values for the authorization ID and other dynamic SQL attributes is called the *dynamic SQL statement behavior*. The four possible behaviors are run, bind, define, and invoke.

This discussion applies to dynamic SQL statements that refer to objects at the current server. For those that refer to objects elsewhere, see "Authorization IDs and remote execution" on page 77.

In addition to determining the authorization ID, DYNAMICRULES also controls other dynamic SQL attributes such as the implicit qualifier that is used for unqualified alias, index, sequence, table, trigger, and view names; the source for application programming options; and whether certain SQL statements can be invoked dynamically.

As the following table shows, the combination of the value of the DYNAMICRULES option and the run time environment determines which of the four SQl statement behavior is used. DYNAMICRULES(RUN), which implies run behavior, is the default.

Table 6. How DYNAMICRULES and the run time environment determine dynamic SQL statement behavior

	Behavior of dynamic SQL statements	
DYNAMICRULES value	**Stand-alone program environment**	**User-defined function or stored procedure environment**
RUN	Run behavior	Run behavior
BIND	Bind behavior	Bind behavior
DEFINERUN	Run behavior	Define behavior
DEFINEBIND	Bind behavior	Define behavior
INVOKERUN	Run behavior	Invoke behavior
INVOKEBIND	Bind behavior	Invoke behavior

Note: BIND and RUN values can be specified for packages, plans, and native SQL procedures. The other values can be specified for packages and native SQL procedures but not for plans.

In the following behavior descriptions, a package that *runs under* a user-defined function or stored procedure package is a package whose associated program meets one of the following conditions:

- The program is called by a user-defined function or stored procedure.

- The program is in a series of nested calls that start with a user-defined function or stored procedure.

Run behavior

DB2 uses the authorization IDs of the application process and the SQL authorization ID (the value of special register CURRENT SQLID) for authorization checking of dynamic SQL statements. If the process is running in a trusted context with a role associated with the primary authorization ID, the authorization IDs of the application process include this role.

A process that uses a plan and its associated packages is called an *application process*. At any time, the SQL authorization ID is the value of CURRENT SQLID. This SQL special register can be initialized by the connection or sign-on exit routine. If the exit routine does not set a value, the initial value of CURRENT SQLID is the primary authorization ID of the process. You can use the SQL statement SET CURRENT SQLID to change the value of CURRENT SQLID. Unless some authorization ID of the process has SYSADM authority, the new value must be one of the authorization IDs of the process. Thus, CURRENT SQLID usually contains either the primary authorization ID of the process or one of its secondary authorization IDs. The CURRENT SQLID cannot contain a role.

Privilege set: If the dynamically prepared statement is other than an ALTER, CREATE, COMMENT, DROP, GRANT, RENAME, or REVOKE statement, each privilege required for the statement can be a privilege designated by any authorization ID of the process. Therefore, the privilege set is the union of the set of privileges held by each authorization ID of the process. When the process is running in a trusted context with a role, the authorization IDs of the process include this role.

If the dynamic SQL statement is an ALTER, CREATE, COMMENT, DROP, GRANT, RENAME, or REVOKE statement, the only authorization ID that is used for authorization checking is the SQL authorization ID. Therefore, the privilege set is the privileges held by that single authorization ID of the process. If the process is running in a trusted context using the ROLE AS OBJECT OWNER clause for the a CREATE, GRANT, or REVOKE statement, the single authorization ID of the process that is checked is the role that is in effect.

Implicit qualification: As explained under "Qualification of unqualified object names" on page 65, when an SQL statement is dynamically prepared, the values of the CURRENT SCHEMA special register is used as the implicit qualifier. For example, it is used as the implicit qualifier for all unqualified tables, aliases, views, indexes, and sequences.

Bind behavior

The same rules that are used to determine the authorization ID for static (embedded) statements are used for dynamic statements. DB2 uses the authorization ID of the owner of the package or plan for authorization checking of dynamic SQL statements, as explained in detail under "Authorization IDs and statement preparation" on page 74.

Privilege set: The privilege set is the privileges that are held by the owner of the package or plan.

Implicit qualification: The identifier specified in the QUALIFIER option of the bind command that is used to bind the SQL statements, or the CREATE PROCEDURE or ALTER PROCEDURE statement that is used to create a version of an SQL procedure is the implicit qualifier for all unqualified

tables, views, aliases, indexes, and sequences. If the QUALIFIER option was not used when the plan, package, or native SQL procedure was created or last changed, the implicit qualifier is the owner of the plan, package, or native SQL procedure.

Define behavior
Define behavior applies only if the dynamic SQL statement is in a package that is run as a stored procedure or user-defined function (or *runs under* a stored procedure or user-defined function package), and the package was bound with DYNAMICRULES(DEFINEBIND) or DYNAMICRULES(DEFINERUN). DB2 uses the authorization ID of the stored procedure or user-defined function owner (the definer) for authorization checking of dynamic SQL statements in the application package.

Privilege set: The privilege set is the privileges that are held by the authorization ID of the owner.

Implicit qualification: The stored procedure or user-defined function owner is also the implicit qualifier. For example, the owner is the implicit qualifier for unqualified table, view, alias, index, and sequence names.

Invoke behavior
Invoke behavior applies only if the dynamic SQL statement is in a package that is run as a stored procedure or user-defined function (or *runs under* a stored procedure or user-defined function package), and the package was bound with DYNAMICRULES(INVOKEBIND) or DYNAMICRULES(INVOKERUN). DB2 uses the stored procedure or user-defined function invoker for authorization checking of dynamic SQL statements in the application package. The invoker can also be a role.

Privilege set: The privilege set is the privileges that are held by the invoker. However, if the invoker is the primary authorization ID of the process or the CURRENT SQLID value, secondary authorization IDs are also checked. This includes the role of the primary authorization ID, if running in a trusted context with a role. In that case, the privilege set is the union of the set of privileges held by each authorization ID of the process.

Implicit qualification: The stored procedure or user-defined function invoker is also the implicit qualifier. For example, it is the implicit qualifier for unqualified table, view, alias, index, and sequence names. The invoker can also be a role.

Restricted statements when run behavior does not apply: When bind, define, or invoke behavior is in effect, you cannot use the following dynamic SQL statements: ALTER, CREATE, COMMENT, DROP, GRANT, RENAME, and REVOKE.

Related reference:

BIND and REBIND options (DB2 Commands)

Privileges required for using dynamic SQL statements (Managing Security)

Authorization IDs and remote execution

The authorization rules for remote execution depend on whether the distributed operation is DRDA access with a DB2 for z/OS server and requester. DRDA access with a server and requester other than DB2 can also effect the authorization rules for remote execution.

DRDA access with DB2 for z/OS only

To prepare and execute SQL statements using DRDA access, certain privileges are required by the package owner and additional privileges are required by the user who invokes the application.

Any static statement executed using DRDA access is in a package bound at a server other than the local DB2 subsystem. Before the package can be bound, its owner must have the BINDADD privilege and the CREATE IN privilege for the package's collection. Also required are enough privileges to execute the package's static SQL statements that refer to data on that server. All these privileges are recorded in the DB2 catalog of the server, not in the catalog of the local DB2 subsystem. Such privileges must be granted by GRANT statements executed at the server. This allows the server to control the creation and use of packages that are run from other DBMSs.

A user who invokes an application that has a plan at the local DB2 subsystem must have the EXECUTE privilege on the plan recorded in the DB2 catalog of the local subsystem. If that application uses a package that is bound at a DB2 server other than the local DB2 requester, the EXECUTE privilege on the package must also be recorded in the DB2 catalog of the server. The ID that must hold the authorization to run the package at the DB2 server depends on the value of the PRIVATE_PROTOCOL subsystem parameter at the DB2 server:

- If PRIVATE_PROTOCOL is set to NO, EXECUTE authority on the package must be explicitly granted to the primary user ID or an associated secondary ID at the DB2 server. If the local requester application invokes a stored procedure that resides at the DB2 server, EXECUTE authority on the stored procedure package must be explicitly granted at the DB2 server to the owner of the package that issues the CALL statement if either of the following is true:
  - The owner of the stored procedure does not have the authority to execute the remote stored procedure package.
  - The CALL statement is in the form of CALL: host-variable and neither the primary user ID nor an associated secondary ID has the authority to execute the remote stored procedure package.
- If PRIVATE_PROTOCOL is not set to NO, EXECUTE authority on the package must be explicitly granted to the local requester plan owner at the DB2 server. The plan owner needs no other privilege to execute the package. If the local requester application invokes a stored procedure that resides at the DB2 server, EXECUTE authority on the stored procedure package must be explicitly granted at the DB2 server to the DB2 requester plan owner of the application that issues the CALL statement if either of the following is true:
  - The owner of the stored procedure does not have the authority to execute the remote stored procedure package.
  - The CALL statement is in the form of CALL: host-variable and neither the primary user ID nor an associated secondary ID has the authority to execute the remote stored procedure package.

EXECUTE authority is also required to use a package for a user-defined function, trigger, or stored procedure that resides at the DB2 server. However, except as previously described for a specific stored procedure case, the PRIVATE_PROTOCOL subsystem parameter is not used to determine the ID that is required to hold the EXECUTE privilege on that package. In the case of trigger packages, the authorization ID of the SQL statement that activates the trigger must have the EXECUTE privilege on the trigger. Again, all these privileges must be recorded in the DB2 catalog of the server.

Having the appropriate privileges recorded as described above allows the execution of the static SQL statements in the package, and the execution of dynamic SQL statements if DYNAMICRULES bind, define, or invoke behavior is in effect. If DYNAMICRULES run behavior is in effect, the authorization rules for dynamic SQL statements is different. Authorization for the execution of dynamic SQL statements must come from the set of authorization IDs that are derived during connection processing, and, if the process is running in a trusted connection, the role that is in effect. An application goes through connection processing when it first connects to a server or when it reuses a CICS or IMS thread that has a different primary authorization ID.

If an application uses Recoverable Resources Manager Services attachment facility (RRSAF) and has no plan, authority to execute the package is determined in the same way as when the requester is not DB2 for z/OS.

Related concepts:

Managing connection requests from local applications (Managing Security)

"DRDA access with a server or requester other than DB2"

Related tasks:

Checking authorization at a DB2 database server (Managing Security)

DRDA access with a server or requester other than DB2

Specific privileges are required depending on whether DB2 is the server or the requester involved in DRDA access.

DB2 for z/OS as the server: If the requester is not a DB2 for z/OS subsystem, there is no DB2 application plan involved. In this case, the privilege set of the authorization ID, which is determined by the DYNAMICRULES behavior, must have the EXECUTE privilege on the package. Dynamic SQL statements in the package are executed according to the DYNAMICRULES behavior, as described in "Authorization IDs and dynamic SQL" on page 75.

DB2 for z/OS as the requester: The authorization rules for remote execution are those of the server.

Authorization ID translations

When certain authorization IDs are sent to a remote DBMS, those authorization IDs might undergo translation before being used.

Translation can occur for a primary authorization ID, the authorization ID of the owner of an application plan, or the authorization ID of the owner of a package. For example, a user known as SMITH at the local DBMS could be known, after translation, as JONES at the server. Likewise, a package owner known as GRAY could be known as WINTERS at the server. If so, JONES or WINTERS would be used, instead of SMITH or GRAY, to determine the authorization ID for dynamic SQL statements in the package. If the DYNAMICRULES run behavior applies, JONES, who is executing the dynamic statement at the server, is used. If DYNAMICRULES bind behavior applies, WINTERS, the package owner at the server, is used.

Two sets of communications database (CDB) catalog tables control the translations. One set is at the local DB2, and the other set is at the remote DB2. Translation can take place at either or both sites.

Related concepts:

Communications database for the requester (Managing Security)

Communications database for the server (Managing Security)

Other security measures

Even if DB2 authority requirements are satisfied, other security measures can be in effect when distributed data is accessed.

The fact that DB2 authority requirements are satisfied does not guarantee that a user has access to a given server. Other security measures can also come into play. For example, requests to execute remote SQL statements could be denied based on Resource Access Control Facility (RACF) considerations. Developing such security measures is discussed in *DB2 Administration Guide*.

Data types

DB2 supports both IBM-supplied data types (built-in data types) and user-defined data types (distinct types).

The smallest unit of data that can be manipulated in SQL is called a *value.* How values are interpreted depends on the data type of their source. The sources of values are:

- Columns
- Constants
- Expressions
- Functions
- Special registers
- Variables (such as host variables, SQL variables, global variables, parameter markers, and parameters of routines)

The following topics describes the built-in data types and distinct types.

Figure 16 on page 81 shows the built-in data types that DB2 supports.

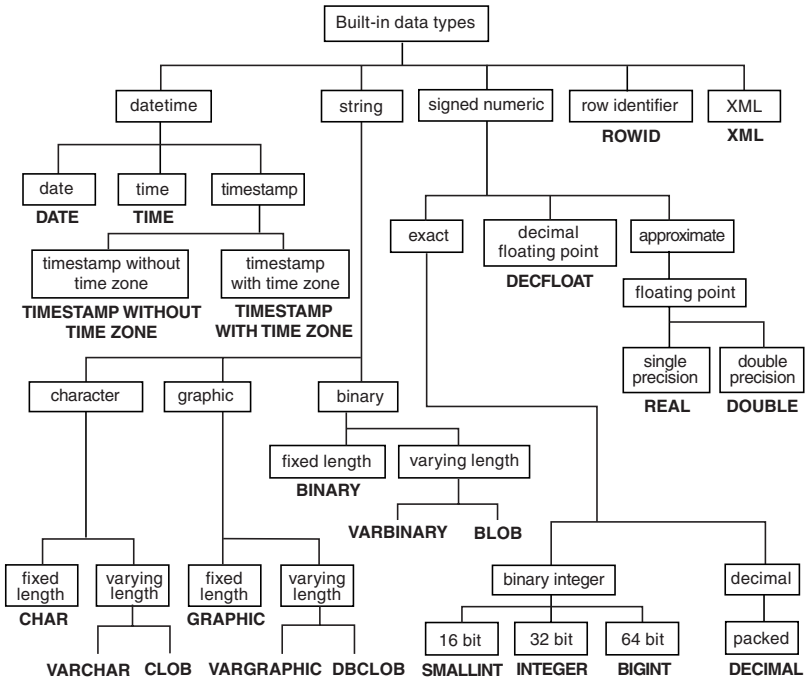


Figure 16. Built-in data types supported by DB2

Nulls

All data types include the null value. Distinct from all nonnull values, the null value is a special value that denotes the absence of a (nonnull) value.

Although all data types include the null value, some sources of values cannot provide the null value. For example, constants, columns that are defined as NOT NULL, and special registers cannot contain null values; the COUNT and COUNT_BIG functions cannot return a null value; and ROWID columns cannot store a null value although a null value can be returned for a ROWID column as the result of a query.

Numbers

The numeric data types are categorized as exact numerics: binary integer and decimal; decimal floating point; and approximate numerics: floating-point

Binary integer includes small integer, large integer, and big integer. Binary numbers are exact representations of integers. Decimal numbers are exact representations of real numbers. Binary and decimal numbers are considered exact numeric types. Decimal floating point numbers include DECFLOAT(16) and DECFLOAT(34), which are capable of representing either 16 or 34 significant digits. Floating-point includes single precision and double precision. Floating-point numbers are approximations of real numbers and are considered approximate numeric types.

All numbers have a sign, a precision, and a scale. If a column value is zero, the sign is positive. Decimal floating point has distinct values for a number and the same number with various exponents (for example: 0.0, 0.00, 0.0E5, 1.0, 1.00, 1.0000). The precision is the total number of binary or decimal digits excluding the

sign. The scale is the total number of binary or decimal digits to the right of the decimal point. If there is no decimal point, the scale is zero.

Small integer (SMALLINT)

A *small integer* is a binary integer with a precision of 15 bits. The range of small integers is -32768 to +32767.

Large integer (INTEGER)

A *large integer* is a binary integer with a precision of 31 bits.

The range of large integers is -2147483648 to +2147483647.

Big integer (BIGINT)

A *big integer* is a binary integer with a precision of 63 bits.

The range of big integers is -9223372036854775808 to +9223372036854775807.

Single precision floating-point (REAL)

A *single precision floating-point* number is a short (32 bits) floating-point number.

The range of single precision floating-point numbers is about -7.2E+75 to 7.2E+75. In this range, the largest negative value is about -5.4E-79, and the smallest positive value is about 5.4E-079.

Double precision floating-point (DOUBLE or FLOAT)

A *double precision floating-point* number is a long (64 bits) floating-point number.

The range of double precision floating-point numbers is about -7.2E+75 to 7.2E+75. In this range, the largest negative value is about -5.4E-79, and the smallest positive value is about 5.4E-079.

Decimal (DECIMAL or NUMERIC)

A *decimal* number is a packed decimal number with an implicit decimal point.

The position of the decimal point is determined by the precision and the scale of the number. The scale, which is the number of digits in the fractional part of the number, cannot be negative or greater than the precision. The maximum precision is 31 digits.

All values of a decimal column have the same precision and scale. The range of a decimal variable or the numbers in a decimal column is *-n* to *+n*, where *n* is the largest positive number that can be represented with the applicable precision and scale. The maximum range is $1 - 10^{31}$ to $10^{31} - 1$.

Decimal floating-point (DECFLOAT)

A *decimal floating-point* value is an IEEE 754r number with a decimal point. The position of the decimal point is stored in each decimal floating-point value.

The maximum precision is 34 digits.

The range of a decimal floating point number is either 16 or 34 digits of precision, and an exponent range of respectively 10^{-383} to 10^{+384} or 10^{-6143} to 10^{+6144}.

In addition to the finite numbers, decimal floating point numbers are able to represent one of the following named special values:

- Infinity - a value that represents a number whose magnitude is infinitely large.

- Quiet NaN - a value that represents undefined results which does not cause an invalid number condition.
- Signaling NaN - a value that represents undefined results which will cause an invalid number condition if used in any numerical operation.

When a number has one of these special values, its coefficient and exponent are undefined. The sign of an infinity is significant (that is, it is possible to have both positive and negative infinity). The sign of a NaN has no meaning for arithmetic operations. INF can be used in place of INFINITY.

Numeric host variables

Numeric host variables can be defined in all languages with a few exceptions.

Binary integer variables can be defined in all host languages.

Floating-point variables can be defined in all host languages. All languages, except Java, support System/390® floating-point format. Assembler, C, C++, PL/I, and Java also support IEEE floating-point format. In assembler, C, C++, and PL/I programs, the SQL processing option FLOAT tells DB2 whether floating-point variables contain data in System/390 floating-point format or IEEE floating-point format. The contents of floating-point host variables must match the format that is specified with the FLOAT SQL processing option.

Decimal variables can be defined in all host languages except Fortran.

In COBOL, decimal numbers can be represented in the following formats:

- Packed decimal format, denoted by USAGE PACKED-DECIMAL or COMP-3
- External decimal format, denoted by USAGE DISPLAY with SIGN LEADING SEPARATE
- NATIONAL decimal format denoted by USAGE NATIONAL and SIGN LEADING SEPARATE

Decimal floating-point variables can be defined in Assembler, C, C++, PL/I, and Java.

String representations of numeric values

String representations of numeric values can be used in some contexts. A valid string representation of a numeric value must conform to the rules for numeric constants.

The encoding scheme in use determines what type of strings can be used for string representation of numeric values. For ASCII and EBCDIC, a string representation of a numeric value must be a character string. For UNICODE, a string representation of a numeric value can be either a character string or a graphic string. Thus, the only time a graphic string can be used for a numeric value is when the encoding scheme is UNICODE.

When a decimal, decimal floating-point, or floating-point number is cast to a string (for example, using a CAST specification), the implicit decimal point is replaced by the default decimal separator character that is in effect when the statement is prepared. When a string is cast to a decimal, decimal floating-point, or floating-point value (for example, using a CAST specification), the default decimal separator character in effect when the statement was prepared is used to interpret the string.

For more information, see “Constants” on page 148.

Subnormal numbers and underflow

The decimal floating-point data type has a set of non-zero numbers that fall outside the range of normal decimal floating-point values. These numbers are called subnormal.

Non-zero numbers whose adjusted exponents are less than E_{min}[4] are called subnormal numbers. These subnormal numbers are accepted as operands for all operations and can result from any operation. If a result is subnormal before any rounding occurs, the subnormal condition is returned.

For a subnormal result, the minimum values of the exponent becomes E_{min} - (precision-1), called E_{tiny}, where precision is the working precision. If necessary, the result will be rounded to ensure that the exponent is no smaller than E_{tiny}. If the result becomes inexact during rounding, an underflow condition is returned. A subnormal result does not always return the underflow condition but will always return the subnormal condition.

When a number underflows to zero during a calculation, its exponent will be E_{tiny}. The maximum value of the exponent is unaffected.

The maximum value of the exponent for subnormal numbers is the same as the minimum value of the exponent which can arise during operations that do not result in subnormal numbers. This occurs where the length of the coefficient in decimal digits is equal to the precision.

Character strings

A *character string* is a sequence of bytes. The length of the string is the number of bytes in the sequence. If the length is zero, the value is called the *empty string*. The empty string should not be confused with the null value.

Default CCSIDs

The value of the field MIXED DATA (on installation panel DSNTIPF) determines the default CCSIDs for a character string.

The following table shows how the value of the field MIXED DATA (on installation panel DSNTIPF) determines the default CCSIDs for a character string.

Table 7. Default CCSIDs for character strings

Encoding scheme	Value of MIXED DATA field	Default attribute
ASCII or EBCDIC	NO	Character: SBCS The value of the ASCII CCSID or EBCDIC CCSID field on installation panel determines the system CCSID for SBCS data.
ASCII or EBCDIC	YES	Character: MIXED The value of the ASCII CCSID or EBCDIC CCSID field on installation panel DSNTIPF determines the system CCSID for SBCS data, MIXED, and graphic data.

Table 7. Default CCSIDs for character strings (continued)

Encoding scheme	Value of MIXED DATA field	Default attribute
Unicode	Not applicable	Character: MIXED The CCSIDs are: • 367 for SBCS data • 1208 for MIXED data • 1200 for graphic data

Fixed-length character strings

When fixed-length character string distinct types, columns, and variables are defined, the length attribute is specified, and all values have the same length. For a fixed-length character string, the length attribute must be between 1 and 255 inclusive.

Varying-length character strings

The types of varying-length character strings are VARCHAR and *character large object (CLOB)*. A CLOB is a type of LOB. A CLOB column is useful for storing large amounts of character data, such as documents written with a single character set.

When varying-length character strings, distinct types, columns, and variables are defined, the maximum length is specified and this length becomes the length attribute except for C NUL-terminated strings. Actual values might have a smaller value. For varying-length character strings, the length specifies the number of bytes.

For a VARCHAR string, the length attribute must be between 1 and 32704. For a VARCHAR column, the maximum for the length attribute is determined by the record size that is associated with the table, as described in Maximum record size the description of the CREATE TABLE statement. For a CLOB string, the length attribute must be between 1 and 2 147 483 647 inclusive. (2 147 483 647 is 2 gigabytes minus 1 byte.) For more information about CLOBs, see "Large objects (LOBs)" on page 96.

Character string variables

Character string variables follow specific rules for use in host languages.

- Fixed-length character string variables can be used in all languages except REXX and Java. In C, CHAR string variables are limited to a length of 1.
- Varying-length character string variables can be used in all host languages with the following exceptions:
  - Fortran: varying-length non-LOB character strings cannot be used.
  - Assembler, C, and COBOL: varying-length non-LOB strings are simulated as described in *DB2 Application Programming and SQL Guide*. In C, NUL-terminated strings can also be used.
  - REXX: CLOBs and DBCLOBs cannot be used.

Character string encoding schemes

The method of representing DBCS and MBCS characters within a mixed string differs among the encoding schemes.

Each character string is further defined as one of the following subtypes:

Bit data

Data that is not associated with a coded character set and, therefore, is never converted. The CCSID for bit data is X'FFFF' (65535). The bytes do not represent characters.

Bit data is a form of character data. The pad character is a blank for assignments to bit data; the pad character is X'00' for assignments to binary data. It is recommended that binary data be used instead of character for bit data.

If both operands in a predicate are EBCDIC, both operands are padded with X'40'. Otherwise, both operands are padded with X'20'. For example, if both operands are ASCII, or if one operand is ASCII and the other operand is EBCDIC, both are padded with X'20'.

SBCS data

Data in which every character is represented by a single byte. Each SBCS string has an associated CCSID. If necessary, an SBCS string is converted before it is used in an operation with a character string that has a different CCSID.

Mixed data

Data that can contain a mixture of characters from a single-byte character set (SBCS) and a multiple-byte character set (MBCS). Each mixed string has an associated CCSID. If necessary, a mixed string is converted before an operation with a character string that has a different CCSID. If a mixed data string contains an MBCS character, it cannot be converted to SBCS data.

EBCDIC mixed data can contain shift characters, which are not MBCS data.

When the encoding scheme is Unicode or the DB2 installation is defined to support mixed data, DB2 recognizes MBCS sequences within mixed data string when performing character sensitive operations. These operations include parsing, character conversion, and the pattern matching specified by the LIKE predicate.

Character strings with a CLOB data type can only be SBCS or MIXED. BLOB should be used for binary strings.

The method of representing DBCS and MBCS characters within a mixed string differs among the encoding schemes.

- ASCII reserves a set of code points for SBCS characters and another set as the first half of DBCS characters. When it encounters the first half of a DBCS character, the system reads the next byte in order to obtain the complete character.
- EBCDIC makes use of two special code points:
  - A shift-out character (X'0E') to introduce a string of DBCS characters.
  - A shift-in character (X'0F') to end a string of DBCS characters.

  DBCS sequences within mixed data strings are recognized as the string is read from left to right. At any time, the reading of the string is in SBCS mode or DBCS mode. In SBCS mode, which is the initial mode, any byte other than a shift-out is interpreted as an SBCS character. When a shift-out is read, the mode switches to DBCS mode. In DBCS mode, the next byte and every second byte after that byte is interpreted as the first byte of a DBCS character unless it is a shift character. If the byte is a shift-out, an error occurs. If the byte is a shift-in, the mode returns to SBCS mode. An error occurs if the mode is still DBCS mode

after processing the last byte of the string. Because of the shift characters, EBCDIC mixed data requires more storage than ASCII mixed data.

- UTF-8 is a varying-length encoding of byte sequences. The high bits indicate the part of the sequence to which a byte belongs. The first byte indicates the number of bytes to follow in a byte sequence.

Examples

The same mixed date character string can be represented as character and hexadecimal data in different encoding schemes.

For the same mixed data character string, the following table shows character and hexadecimal representations of the character string in different encoding schemes. In EBCDIC, the shift-out and shift-in characters are needed to delineate the double-byte characters.

Table 8. Example of a character string in different encoding schemes

Data type and encoding scheme	Character representation	Hexadecimal representation (with spaces separating each character)
9 bytes in ASCII	元 gen 気 ki	8CB3 67 65 6E 8B43 6B 69
13 bytes in EBCDIC	SO 元 SI gen SO 気 SI ki	0E 4695 0F 87 85 95 0E 45B9 0F 92 89
11 bytes in Unicode UTF-8	元 gen 気 ki	E58583 67 65 6E E6B097 6B 69

Because of the differences of the representation of mixed data strings in ASCII, EBCDIC, and Unicode, mixed data is not transparently portable. To minimize the effects of these differences, use varying-length strings in applications that require mixed data and operate on ASCII, EBCDIC, and Unicode data.

String unit specifications

The ability to specify string units for certain built-in functions and on the CAST specification allows you to process string data in a more "character-based manner" than a "byte-based manner". The string unit determines the length in which the operation is to occur. You can specify CODEUNITS32, CODEUNITS16, or OCTETS as the units for the operation.

CODEUNITS32
Specifies that Unicode UTF-32 is the units for the operation. CODEUNITS32 is useful when an application wants to process data in a simple fixed-length format and needs the same answer regardless of the storage format of the data (ASCII, EBCDIC, UTF-8, or UTF-16). Although the answers are in terms of CODEUNITS32, the data is not converted to UTF-32 to perform the function.

CODEUNITS16
Specifies that Unicode UTF-16 is the units for the operation. CODEUNITS16 is useful when an application wants to know how many double-byte characters are in a string.

OCTETS
Specifies that bytes are the units for the operation. OCTETS is often used when an application is interested in allocation buffer space or when operations need to use simple byte processing.

Determining the length of a string by counting in string units (CODEUNITS16 or CODEUNITS32) or bytes (OCTETS) can result in different answers. When OCTETS is specified, the length of a string is determined by simply counting the number of bytes in the string. Counting by CODEUNITS16 or CODEUNITS32 gives the same answer unless the data contains supplementary characters. For information about the difference between CODEUNITS16 and CODEUNITS32 when the data contains supplementary characters, see "Difference between CODEUNITS16 and CODEUNITS32" on page 89.

Example: Assume that NAME is a VARCHAR(128) column, encoded in Unicode UTF-8, that contains the value 'Jürgen'. The first two queries, which count the length of the string in CODEUNITS32 and CODEUNITS16, returns the same value, 6. The third query, which counts the length of the string in OCTETS, returns the value 7. These values are the length of the string as expressed in the string units that are specified.

```
SELECT CHARACTER_LENGTH(NAME,CODEUNITS32)
  FROM T1 WHERE NAME = 'Jürgen';
SELECT CHARACTER_LENGTH(NAME,CODEUNITS16)
  FROM T1 WHERE NAME = 'Jürgen';
SELECT CHARACTER_LENGTH(NAME,OCTETS)
  FROM T1 WHERE NAME = 'Jürgen';
```

The following table shows the UTF-8, UTF-16, and UTF-32 representations of 'Jürgen'.

Format	Representation of the name 'Jürgen'
UTF-8	x'4A**C3BC**7267656E'
UTF-16	x'004A**00FC**007200670065006E'
UTF-32	x'0000004A**000000FC**00000072000000670000006500000006E'

The bold highlighting in the table demonstrates how the representation of the character *ü* in 'Jürgen' differs between the three string units:

- The UTF-8 representation of the character ü is X'C3BC'. In UTF-8, characters that are not in the Latin-1 subset (essentially a through z, A through Z, and 0 through 9), such as accented characters or Japanese characters, are represented by multiple bytes.
- The UTF-16 representation of the character ü is X'00FC'. In UTF-16, each character is represented in 2 bytes. UTF-16 supplementary characters take two 2-byte code points.
- The UTF-32 representation of the character ü is X'000000FC'. In UTF-32, each character is represented in 4 bytes.

Specifying the string units on a built-in function does not affect the data type or the CCSID of the result of the function. If necessary, DB2 converts the data to Unicode for evaluation when CODEUNITS32 or CODEUNITS16 is specified. DB2 always evaluates the data in the encoding scheme of the output data when OCTETS is specified. For more information about the data types and CCSIDs of the results of functions, see the description of each function.

Differences between the way that characters are represented in ASCII, EBCDIC, and Unicode can affect the results of your queries.

Example: Assume that NAME is a VARCHAR(128) column, encoded in EBCDIC (CCSID 37), that contains the value 'Mit freundlichen Grüßen, Jürgen'. The following query returns the string 'Mit freundlichen Grüß':

```
SELECT SUBSTRING(C1,1,21,CODEUNITS16)
  FROM T1 WHERE C1 = 'Mit freundlichen Grüßen, Jürgen';
```

The following table shows the result data in more detail:

Format	Representation of 'Mit freundlichen Grüß'
EBCDIC	D489A340869985A49584938983888595 40C799**DC59**
UTF-8	4D697420667265756E646C696368656E204772**C3BCC39F**
UTF-16	004D006900740020006600720065007500 6E0064006C0069006300680 065006E002000470072**00FC00DF**

The bold highlighting in the table shows that the representation of the characters *ü* and *ß* in UTF-8 and UTF-16 each require two bytes. If OCTETS had been specified on the SUBSTRING function to have the string evaluated in UTF-8 bytes instead of EBCDIC OCTETS or CODEUNITS16, the result would have been 'Mit freundlichen Grü'. The character ß would have been lost.

Difference between CODEUNITS16 and CODEUNITS32:

CODEUNITS16 and CODEUNITS32 return the same answer unless the data contains supplementary characters.

A supplementary character is represented as two UTF-16 code units or one UTF-32 code unit. In UTF-8, a non-supplementary character is represented by 1 to 3 bytes and a supplementary character is represented by 4 bytes. In UTF-16, a non-supplementary character is represented by one CODEUNIT16 code unit or 2 bytes, and a supplementary character is represented by two CODEUNIT16 code units or 4 bytes. In UTF-32, a character is represented by one CODEUNIT32 code unit or 4 bytes. Thus, CODEUNITS16 and CODEUNITS32 return different answers when the data contains supplementary characters.

Example 1: The following table shows the hexadecimal values for the mathematical bold capital A and the Latin capital letter A. The mathematical bold capital A is a supplementary character that is represented by 4 bytes in UTF-8, UTF-16, and UTF-32.

Character	UTF-8 representation	UTF-16 representation	UTF-32 representation
Unicode value \u1D400 - 'A' MATHEMATICAL BOLD CAPITAL A	X'F09D9080'	X'D835DC00'	X'0001D400'
Unicode value \u0041 - 'A' LATIN CAPITAL LETTER A	X'41'	X'0041'	X'00000041'

Assume that C1 is a VARCHAR(128) column, encoded in Unicode UTF-8, and that table T1 contains one row with the value of the mathematical bold capital A (X'F09D9080'). The following similar queries return different answers:

```
-- Query:                                          -- Returns the value:
SELECT CHARACTER_LENGTH(C1,CODEUNITS32) FROM T1;    -- 1
SELECT CHARACTER_LENGTH(C1,CODEUNITS16) FROM T1;    -- 2
SELECT CHARACTER_LENGTH(C1,OCTETS) FROM T1;         -- 4
```

Example 2: Assume that C1 is a VARCHAR(128) column, encoded in Unicode UTF-8, and that table T1 contains one row with the value of the mathematical bold capital A (X'F09D9080'). The following similar queries return different answers.

```
-- Query:                                              -- Returns the value:
SELECT HEX(SUBSTRING(C1,1,1,CODEUNITS32)) FROM T1;     -- X'F09D9080'
SELECT HEX(SUBSTRING(C1,1,1,CODEUNITS16)) FROM T1;     -- X'20'
SELECT HEX(SUBSTRING(C1,1,2,CODEUNITS16)) FROM T1;     -- X'F09D9080'
SELECT HEX(SUBSTRING(C1,1,1,OCTETS)) FROM T1;          -- X'20'
SELECT HEX(SUBSTR(C1,1,1)) FROM T1;                    -- X'F0'
```

The value X'20' is the pad (blank) character.

Determining the length attribute of the final result:

When CODEUNITS32, CODEUNITS16, or OCTETS is specified for a function or the CAST specification, the length attribute of the final result string is calculated by applying specific formulas depending on which function is specified.

To determine the final result of a function or the CAST specification, DB2 might need to use an intermediate result string if CODEUNITS32 or CODEUNITS16 is specified, depending on the encoding scheme of the data:

- ASCII and EBCDIC data require the use of a UTF-16 intermediate result string when either CODEUNITS32 or CODEUNITS16 is specified.
- UTF-8 data requires the use of a UTF-16 intermediate result string only when CODEUNITS16 is specified.

Regardless of whether an intermediate string is used, when CODEUNITS32, CODEUNITS16, or OCTETS is specified for a function or the CAST specification, the length attribute of the final result string is calculated by applying the formulas that are described in the following table. The length attributes that are calculated at each step in the formulas are measured in bytes, unless indicated otherwise.

Table 9. Formulas for the length attribute of the final result string

Function	Determination of the length attribute of the string[1]

CAST specification
CHAR
CLOB
DBCLOB
GRAPHIC
VARCHAR
VARGRAPHIC
	Follow these three steps to determine the length attribute of the final result:
	1. Length of the intermediate string (IML)
	When CODEUNITS32 or CODEUNITS16 is specified:
	• If the source string is not in Unicode CCSID 1200, 1208, or 367, convert the source string to CCSID 1200, using the formulas in Table 30 on page 142 to determine the result length of the intermediate string (IML).
	• If source string is in Unicode CCSID 1208 or 367, and CODEUNITS16 is specified, convert the source string to CCSID 1200, using the formulas in Table 30 on page 142 to determine the result length of the intermediate string (IML).
	• Otherwise, the intermediate string is the same as the source string.
	When OCTETS is specified:
	• If the CCSID of the source string is different from the CCSID of the result of the function, convert the source string to the CCSID of the result of the function, using the formulas in Table 30 on page 142 to determine the result length of the intermediate string (IML).
	• Otherwise, the intermediate string is the same as the source string.
	Exception: For the GRAPHIC and VARGRAPHIC function, if the source string is EBCDIC, the source is widened with prefix X'42' before the source string is converted to CCSID 1200 and the length of the intermediate string is determined.
	2. Result length attribute of the intermediate string (rl)
	The result length (rl) of the intermediate string depends on whether a *length* argument was explicitly specified.
	If *length* was not specified, the result length (rl) attribute is:
	`rl = IML`
	If *length* was specified, the result length (rl) attribute is:
	`IF (ol * n) < r_IML THEN`
	`rl = ol * n`
	`ELSE`
	`IF intermediate string is in CCSID 1200`
	`(UTF-16) THEN`
	`rl = MIN( ol * n , IML + ( r * 2 ))`
	`ELSE`
	`rl = MIN( ol * n , IML + r )`
	Where:
	• ol = original *length* argument, expressed in the specified string units
	n = 4 bytes for CODEUNITS32
	2 bytes for CODEUNITS16
	• IML = length of the intermediate string
	• r_IML = IML rounded up to next multiple of n
	• r = ol - (r_IML/n), expressed in the specified string units
	The calculation for `r` is an estimate of the difference between the *length* argument and the estimated number of characters of the input argument, expressed in the specified string units.
	3. Length of the final result string (the result of the function)
	The result length attribute of the final string is determined by converting the result length (rl) of the intermediate string to the CCSID of the result of the function, using the formulas in Table 30 on page 142, if CCSID conversion is necessary. Otherwise, the result length attribute of the final string is rl.

Table 9. Formulas for the length attribute of the final result string (continued)

Function	Determination of the length attribute of the string[1]
CHARACTER_ LENGTH LOCATE LOCATE_ IN_STRING POSITION	Follow these three steps to determine the length attribute of the final result: **1. Length of the intermediate string (IML)** The length of the intermediate string (IML) is determined the same way as for the CAST specification. (See Length of the intermediate string (IML).) For the LOCATE, LOCATE_IN_STRING, and POSITION functions, this applies to both the *source-string* and *search-string*. If the CCSIDs of intermediate strings for the converted *source-string* and *search-string* differ, the intermediate string for the *search-string* is converted to the CCSID of intermediate string for the *source-string*. **2. Result length attribute of the intermediate string (rl)** The result length (rl) attribute is always 4 (the length of an integer): `rl = 4` **3. Length of the final result string (the result the function)** The length of the final result of the function is always an integer.

INSERT
OVERLAY
	Follow these three steps to determine the length attribute of the final result:
	1. Length of the intermediate string (IML)
	The length of the intermediate string (IML) for both the *source-string* and the *insert-string* is determined the same way as for the CAST specification. (See Length of the intermediate string (IML).)
	If the CCSIDs of the intermediate strings for the converted *source-string* and *insert-string* differ, the intermediate string for the *insert-string* is converted to the CCSID of the intermediate string for the *source-string*.
	2. Result length attribute of the intermediate string (rl)
	The result length (rl) attribute of the intermediate string depends on whether the start and length arguments are constants.
	If the *start* and *length* arguments are both constants, the result length attribute is:
	`rl = L1 - MIN ( MAX ( 0, L1 - (V2 - 1) * n ), V3 * m) + L4`
	If at least one argument (the *start* or *length* argument) is not a constant, the result length attribute is:
	`rl = L1 + L4`
	Where:
	• L1 and L4 are the length attributes of the intermediate strings of the *source-string* and *insert-string*, respectively.
	• V2 and V3 are the *start* and *length* values, respectively, expressed in the specified string units.
	m= 1 if the intermediate string of the *source-string* is not CCSID 1200 (UTF-16)
	2 if the intermediate string of the *source-string* is CCSID 1200 (UTF-16)
	n= 4 bytes for CODEUNITS32
	2 bytes for CODEUNITS16
	3. Length of the final result string (the result the function)
	The length of the final result is the same as the length of the final result for the CAST specification. (See Length attribute of the final result string (the result of the function).)

Table 9. Formulas for the length attribute of the final result string (continued)

Function	Determination of the length attribute of the string[1]

LEFT
 	
RIGHT
	Follow these three steps to determine the length attribute of the final result:
	1. Length of the intermediate string (IML)
	The length of the intermediate string (IML) is determined the same way as for the CAST specification. (See Length of the intermediate string (IML).)
	2. Result length attribute of the intermediate string (rl)
	The result length (rl) attribute is the same as the length of the intermediate string:
	`rl = IML`
	3. Length of the final result string (the result of the function)
	The result length attribute of the final string is determined by converting the result length (rl) of the intermediate string to the CCSID of the result of the function, using the formulas in Table 30 on page 142, if CCSID conversion is necessary. Otherwise, the result length attribute of the final string is rl.
	The result length attribute of the final string is:
	`MIN(length of source string, length of CCSID converted string )`

SUBSTRING	Follow these three steps to determine the length attribute of the final result:
	1. Length of the intermediate string (IML)
	The length of the intermediate string (IML) is determined the same way as for the CAST specification. (See Length of the intermediate string (IML).)
	2. Result length attribute of the intermediate string (rl)
	The result length (rl) of the intermediate string depends on whether a *length* argument was explicitly specified.
	If *length* was not specified, the result length (rl) attribute is:
	`rl = IML`
	If *length* was specified, the result length (rl) attribute is:
	`rl = MIN(ol * n, IML)`
	Where:
	• ol = original *length* argument, expressed in the specified string units
	n = 4 bytes for CODEUNITS32
	2 bytes for CODEUNITS16
	• IML = length of the intermediate string
	3. Length of the final result string (the result of the function)
	The length of the final result string is the same as for LEFT built-in function.

Note:

1. The final value of the calculation for each length attribute (IML, rl, and the final result of the function) is limited by the maximum length of the function or by the maximum length of the corresponding data type of the result, whichever is applicable. Each length attribute is expressed in terms of bytes.

Example 1: Assume that T1 is a table encoded in EBCDIC and C1 is a CHAR(26) column (SBCS data with EBCDIC CCSID 37). The CHAR function is invoked in the following statement:

```
SELECT CHAR(C1,10,CODEUNITS32) as COL1 FROM T1;
```

DB2 uses an intermediate string to evaluate the function and determines the intermediate and final result string lengths using these steps:

1. C1, which is SBCS EBCDIC 37 data, is converted to Unicode 1200 (UTF-16). The result length of the conversion (using the formula from Table 30 on page 142, X * 2) is 26 * 2. Thus, the length of the intermediate string is 52 bytes (IML = 52).
2. The CHAR function is evaluated against the first 10 UTF-32 characters in this string. The result length attribute is 40 bytes (`rl = ol * n or 10 * 4`) because `ol * n < r_IML or 40 < 52`.
3. The 40 bytes of the string are converted back to SBCS EBCDIC 37. The result length of the conversion (using the formula from Table 30 on page 142, X * .5) is 40 * .5. Thus, the length of the final result of the functions is 20 bytes.

Example 2: This example is similar to the first example, except that the specified length for the function is 20 instead of 10. Assume that T1 is a table encoded in EBCDIC and C1 is a CHAR(26) column (SBCS data with EBCDIC CCSID 37). The CHAR function is invoked in the following statement:

```
SELECT CHAR(C1,20,CODEUNITS32) as COL1 FROM T1;
```

DB2 uses an intermediate string to evaluate the function and determines the intermediate and final result string lengths using these steps:

1. C1, which is SBCS EBCDIC 37 data, is converted to Unicode 1200 (UTF-16). The result length of the conversion (using the formula from Table 30 on page 142, X * 2) is 26 * 2. Thus, the length of the intermediate result string is 52 bytes (IML = 52).
2. The CHAR function is evaluated against the first 20 UTF-32 characters in this intermediate string. However, because the estimated number of characters in the intermediate string, as expressed in the specified string units, is only 13 characters (`r_IML/n or 52/4`), the intermediate string must be padded with 7 padding characters to satisfy the 20 characters that are requested (`r = ol - (r_IML/n) or 20 - 13`). In Unicode 1200 (UTF-16), each padding character takes 2 bytes.

   The result length attribute is then calculated to be 66 bytes (`rl = MIN(ol * n, IML + (r * 2)) or MIN(20 * 4, 52 + 14))` because `ol * n < r_IML or 80 < 52` is not true.
3. The 66 bytes of the string are converted back to SBCS EBCDIC 37. The result length of the conversion (using the formula from Table 30 on page 142, X * .5) is 66 * .5. Thus, the length of the final result of the function is 33 bytes.

Graphic strings

A *graphic string* is a sequence of double-byte characters.

The length of the string is the number of characters in the sequence. Like character strings, graphic strings can be empty. An empty string should not be confused with the null value.

Fixed-length graphic strings

When fixed-length graphic string distinct types, columns, and variables are defined, the length attribute is specified and all values have the same length. For a fixed-length graphic string, the length attribute must be between 1 and 127 inclusive. A fixed-length graphic string column can also be called a GRAPHIC column.

Varying-length graphic strings

The types of varying-length graphic strings are VARGRAPHIC and *double-byte character large object (DBCLOB).* DBCLOB is a type of LOB. A DBCLOB column is

useful for storing large amounts of double-byte character data, such as documents written with a single double-byte character set.

When varying-length graphic strings, distinct types, columns, and variables are defined, the maximum length is specified and this length becomes the length attribute. Actual values might have a smaller value. For a varying-length graphic string, the length attribute must between 1 and 16352.

For a varying-length graphic string column, the maximum for the length attribute is determined by the record size associated with the table, as described Maximum record size in the description of the CREATE TABLE statement. For a DBCLOB string, the length attribute must be between 1 and 1 073 741 823 inclusive. In UTF-16, although supplementary characters use two 2-byte code points, supplementary characters are still considered double-byte characters. For more information about DBCLOBs, see "Large objects (LOBs)" on page 96.

Graphic string variables

Graphic string variables must follow certain rules.

Variables with a graphic string type cannot be defined in Fortran. In addition, graphic string variables follow these rules:

- Fixed-length graphic string host variables can be defined in all host languages, except REXX and Java. In C, fixed-length graphic-string variables are limited to a length of 1.
- Varying-length graphic string variables can be defined in all host languages, with the exception of DBCLOBs which cannot be used in REXX.

Graphic string encoding schemes

Each graphic string can be further defined as either double-byte data or Unicode data.

Double-byte data
: Data in which every character is represented by a character from the double-byte character set (DBCS) that does not include shift-out or shift-in characters. Each double-byte graphic string has an associated ASCII or EBCDIC CCSID.

Unicode data
: Data that contains characters represented by two bytes, except supplementary characters, which take two 2-byte code points per character. Each Unicode graphic string is encoded using UTF-16. The CCSID for UTF-16 is 1200.

String units in built-in functions

When working with graphic strings, you can specify the string unit in which the operation is to take place for certain built-in functions and the CAST specification. The string unit determines the length in which the operation is to occur.

For more information about string units, see "String unit specifications" on page 87.

Binary strings

A *binary string* is a sequence of bytes.

The length of a binary string is the number of bytes in the sequence. Binary strings are not associated with any CCSID. There are three binary string data types: BINARY, VARBINARY (BINARY VARYING) and BLOB (BINARY LARGE OBJECT).

Fixed-length binary strings

The type of fixed-length binary strings is BINARY. When fixed-length binary string distinct types, columns, and variables are defined, the length attribute is specified, and all values have the same length. For a fixed-length binary string, the length attribute must be between 1 and 255 inclusive.

Varying-length binary strings

The types of varying-length binary strings are VARBINARY (BINARY VARYING) and BLOB (BINARY LARGE OBJECT)

When varying-length binary strings, distinct types, columns, and variables are defined, the maximum length is specified and this length becomes the length attribute. Actual length values might have a smaller value than the length attribute value. For varying-length binary strings, the actual length specifies the number of bytes in the string.

For a VARBINARY string, the length attribute must be between 1 and 32704. For a VARBINARY string column, the maximum for the length attribute is determined by the record size that is associated with the table, as described in "Maximum record size" on the description of the CREATE TABLE statement. Like a varying-length character string, varying-length binary string could be an empty string.

A binary string column is useful for storing non-character data, such as encoded or compressed data, pictures, voice, and mixed media. Another use is to hold structured data for exploitation by distinct types, user-defined functions, and stored procedures. Note, that although binary strings and FOR BIT DATA character strings might be used for similar purposes, the two data types are not compatible. The BINARY, BLOB, VARBINARY built-in functions and CAST specification can be used to change a FOR BIT DATA character string into a binary string.

Large objects (LOBs)

The term *large object (LOB)* refers to any of the following data types: CLOB, DBCLOB, or BLOB.

CLOB A *character large object (CLOB)* is a varying-length string with a maximum length of 2 147 483 647 bytes (2 gigabytes minus 1 byte). A CLOB is designed to store large SBCS data or mixed data, such as lengthy documents. For example, you can store information such as an employee resume, the script of a play, or the text of novel in a CLOB. Alternatively, you can store such information in UTF-8 in a mixed CLOB. A CLOB is a varying-length character string.

DBCLOB
A *double-byte character large object (DBCLOB)* is a varying-length string with a maximum length of 1 073 741 823 double-byte characters. A DBCLOB is designed to store large DBCS data. For example, you could store the information mentioned for CLOB (an employee resume, the script for a play, or the text of a novel) in UTF-16 in a DBCLOB. A DBCLOB is a varying-length graphic string.

BLOB A *binary large object (BLOB)* is a varying-length string with a maximum

length of 2 147 483 647 bytes (2 gigabytes minus 1 byte). A BLOB is designed to store non-traditional data such as pictures, voice, and mixed media. BLOBs can also store structured data for use by distinct types and user-defined functions. A BLOB is a binary string.

Although BLOB strings and FOR BIT DATA character strings might be used for similar purposes, the two data types are not compatible. The BLOB function can be used to change a FOR BIT DATA character string into a BLOB string.

Restrictions using LOBs

With a few exceptions, you can use LOBs in the same contexts in which you can use other varying-length strings.

The following table shows the contexts in which LOBs cannot be used.

Table 10. Contexts in which LOBs cannot be used

Context of usage	LOB (CLOB, DBCLOB, or BLOB)
A GROUP BY clause	Not allowed
An ORDER BY clause	Not allowed
A CREATE INDEX statement that creates an index using an expression	Not allowed except when the index is created using an expression, in which case an inline LOB column can be referenced as the source data type for the SUBSTR ad SUBSTRING built-in functions.
A SELECT DISTINCT statement	Not allowed
A MERGE statement	Cannot be used in the context of an INCLUDE *column-name* clause
A subselect of a set operation except UNION ALL	Not allowed
Predicates	Cannot be used in any predicate except EXISTS, LIKE, and NULL. This restriction includes a *simple-when-clause* in a CASE expression. *expression* WHEN *expression* in a *simple-when-clause* is equivalent to a predicate with *expression=expression*.
The definition of primary, unique, and foreign keys	Not allowed
Check constraints	Not allowed

Manipulating LOBs using locators

A LOB locator is a host variable with a value that represents a single LOB value in the database server. LOB locators provide a mechanism for you to easily manipulate very large objects in application programs without having to store the entire LOB value on the client machine where the application program might be running.

Because LOB values can be very large, the transfer of these values from the database server to host variables in client application programs can be time consuming. Also, application programs typically process LOB values a piece at a time, rather than as a whole. For these cases, the application can use a *large object locator* (LOB locator) to reference the LOB value.

For example, when selecting a LOB value, an application program could handle the value in either of these two ways:

- Select the entire LOB value and place it into an equally large host variable. This method is acceptable if the application program is going to process the entire LOB value at once.
- Select the LOB value into a LOB locator. Then, using the LOB locator, the application program can issue subsequent database operations on the LOB value (such as using it as a parameter to the scalar functions SUBSTR, CONCAT, COALESCE, LENGTH, doing an assignment, searching the LOB value with LIKE or POSSTR, or using it as a parameter to a user-defined function or procedure) by supplying the LOB locator value as input. The resulting output of the LOB locator operation, for example, the amount of data that is assigned to a client host variable, would then typically be a small subset of the input LOB value.

LOB locators can also represent more than just base values; they can also represent the value associated with a LOB expression. For example, a LOB locator might represent the value associated with:

```
SUBSTR(lob_value_1 CONCAT lob_value_2 CONCAT lob_value_3 , 42, 6000000)
```

For non-locator-based host variables in an application program, when a null value is selected into that host variable, the indicator variable is set to -1, signifying that the value is null. For LOB locators, however, the meaning of indicator variables is slightly different. Because a LOB locator host variable itself can never be null, a negative indicator variable value indicates that the LOB value represented by the LOB locator is null. The null information is kept local to the client by virtue of the indicator variable value (the server does not track null values with valid LOB locators).

A LOB locator represents a value, not a row or location in the database. Therefore, after a value is selected into a LOB locator, no action that is subsequently performed on the original row or table will affect the value that is referenced by the LOB locator. The value associated with a LOB locator is valid until the transaction ends, or until the LOB locator is explicitly freed, whichever comes first.

A LOB locator is also not a database type, and it is never stored in the database. As a result, it cannot participate in views or check constraints. However, values for the SQLTYPE field of the SQLDA exist for LOB locators so that they can be described within an SQLDA structure that is used by FETCH, OPEN, CALL and EXECUTE statements.

For more information about manipulating LOBs with LOB locators, see *DB2 Application Programming and SQL Guide*.

Datetime values

Datetime values are neither strings nor numbers. Nevertheless, datetime values can be used in certain arithmetic and string operations and are compatible with certain strings.

Moreover, strings can represent datetime values, as discussed in "String representations of datetime values" on page 101.

Date

A *date* is a three-part value (year, month, and day) designating a point in time using the Gregorian calendar, which is assumed to have been in effect from the year 1 A.D.

[4] The range of the year part is 0001 to 9999. The range of the month part is 1 to 12. The range of the day part is 1 to 28, 29, 30, or 31, depending on the month and year.

The internal representation of a date is a string of 4 bytes. Each byte consists of two packed decimal digits. The first 2 bytes represent the year, the third byte the month, and the last byte the day.

The length of a DATE column as described in the catalog is the internal length, which is 4 bytes. The length of a DATE column as described in the SQLDA is the external length, which is 10 bytes unless a date exit routine was specified when your DB2 subsystem was installed. (Writing a date exit routine is described in *DB2 Administration Guide*.) In that case, the string format of a date can be up to 255 bytes in length. Accordingly, DCLGEN[5] defines fixed-length string variables for DATE columns with a length equal to the value of the field LOCAL DATE LENGTH on installation panel DSNTIP4, or a length of 10 bytes if a value for the field was not specified.

A character-string representation must have an actual length that is not greater than 255 bytes and must not be a CLOB or DBCLOB.

Time

A *time* is a three-part value (hour, minute, and second) designating a time of day using a 24-hour clock. The range of the hour part is 0 to 24. The range of the minute and second parts is 0 to 59. If the hour is 24, the minute and second parts are both zero.

The internal representation of a time is a string of 3 bytes. Each byte consists of two packed decimal digits. The first byte represents the hour, the second byte the minute, and the last byte the second.

The length of a TIME column as described in the catalog is the internal length which is 3 bytes. The length of a TIME column as described in the SQLDA is the external length which is 8 bytes unless a time exit routine was specified when the DB2 subsystem was installed. (Writing a time exit routine is described in *DB2 Administration Guide*.) In that case, the string format of a time can be up to 255 bytes in length. Accordingly, DCLGEN[5] defines fixed-length string variables for TIME columns with a length equal to the value of the field LOCAL TIME LENGTH on installation panel DSNTIP4, or a length of 8 bytes if a value for the field was not specified.

A character-string representation must have an actual length that is not greater than 255 bytes and must not be a CLOB or DBCLOB.

Timestamp

A *timestamp* is a six-part or seven-part value (year, month, day, hour, minute, second, and optional fractional second) with an optional time zone specification, that represents a date and time.

The time portion of a timestamp value can includes a specification of fractional seconds. The number of digits in the fractional seconds portion is specified using

---

4. Historical dates do not always follow the Gregorian calendar. Dates between 1582-10-04 and 1582-10-15 are accepted as valid dates although they never existed in the Gregorian calendar.

5. DCLGEN is a DB2 DSN subcommand for generating table declarations for designated tables or views. The declarations are stored in z/OS data sets, for later inclusion in DB2 source programs.

an attribute in the range from 0 to 12 with a default of 6. The time zone is the difference in hours and minutes between local time and UTC. The range of the hour offset is -12 to 14, and the minute offset is 00 to 59. The optional time zone is specified in the format ±th.tm, with values ranging from -12.59 to +14.00. A timestamp data type is TIMESTAMP WITHOUT TIME ZONE (generically referred to as TIMESTAMP) or TIMESTAMP WITH TIME ZONE.

TIMESTAMP WITHOUT TIME ZONE

The internal representation of a timestamp is a string of 7 to 13 bytes, each of which consists of two packed decimal digits. The first 4 bytes represent the date, the next 3 bytes the time, and the remaining bytes the fractional seconds based on the precision of the timestamp.

The length of a TIMESTAMP WITHOUT TIME ZONE column as described in the catalog is the internal length, which is 7 to 13 bytes.

The length of a TIMESTAMP WITHOUT TIME ZONE column as described in the SQLDA is between 19 and 32 bytes, which corresponds to the length for the character-string representation of the value. For example, a 19 byte character-string representation has no fractional seconds; a 26 byte character-string representation has 6 digits of fractional seconds; and a 29 byte character-string representation has 9 digits of fractional seconds.

A character-string representation must have an actual length that is not greater than 255 bytes and must not be a CLOB or DBCLOB.

TIMESTAMP WITH TIME ZONE

The external representation of a TIMESTAMP WITH TIME ZONE value is the local timestamp followed by the time zone offset. For example, New York is 5 hours behind London during standard time, so New York time "8:15" on 2010-02-10 can be represented as '2010-02-10-08.15.00-5:00'. This timestamp with time zone value represents a UTC value '2010-02-10-13.15.00', which is derived by subtracting the time zone offset from local timestamp.

The internal representation of a timestamp is a string of 9 to 15 bytes that contains the UTC timestamp followed by the time zone. Each byte consists of 2 packed decimal digits. The first byte consists of two packed decimal digits representing time zone hour and the first bit is used to represent the sign of the time zone offset. The second byte of time zone, representing the time zone minute, also consists of two packed decimal digits. For example, time zone "-3:30" is represented as X'8330' and time zone "5:30" is represented as X'0530'.

The length of a TIMESTAMP WITH TIME ZONE column as described in the catalog is the internal length, which is between 9 to 15 bytes (a 7 to 13 bytes timestamp followed by 2 bytes time zone).

The length of a TIMESTAMP WITH TIME ZONE column as described in the SQLDA is the external length, which is between 147 and 160 bytes and corresponds to the length for the character-string representation of the value. For example, a 147 byte character representation has no fractional seconds, and a 160 byte character-string representation has 12 digits of fractional seconds, where the time zone component is 7 bytes.

A character-string representation must have an actual length that is not greater than 255 bytes and must not be a CLOB or DBCLOB. DCLGEN therefore defines 147 to 160 byte, varying-length string variables for TIMESTAMP WITH TIME ZONE columns.

Datetime host variables

Character-string host variables are normally used to contain date, time, and timestamp values. However, date, time, and timestamp host variables can also be specified in Java as java.sql.Date, java.sql.Time, and java.sql.Timestamp, respectively.

String representations of datetime values

Dates, times, and timestamp values can be represented by strings. For many host languages, there are no special SQL constants for datetime values and, except for Java, no host variables with a data type of date, time, or timestamp. Thus, to be retrieved, a datetime value must be assigned to a string variable.

Values whose data types are DATE, TIME, TIMESTAMP WITHOUT TIME ZONE, or TIMESTAMP WITH TIME ZONE are represented in a form that is transparent to the user of SQL. Dates, times, and timestamps (with or without time zones) can also be represented by strings. These representations directly concern the SQL user because, for many host languages there are no special SQL constants or host variables with a data type for DATE, TIME, TIMESTAMP WITHOUT TIME ZONE, or TIMESTAMP WITH TIME ZONE values (for variables with Java). Thus, to be retrieved, a datetime value must be assigned to a string variable. The format of the resulting string depends on the default date format and the default time format that is in effect when the statement is prepared.

Each datetime value is assigned an encoding scheme. This encoding scheme is used when the datetime value is converted from its internal form to the string representation in the form of the mixed CCSID if the field MIXED DATA is YES on installation panel DSNTIPF. Otherwise the SBCS CCSID of the assigned encoding scheme is used. For Unicode, the mixed CCSID is always used. The following table shows how the encoding scheme is determined:

Table 11. The encoding scheme of datetime values

Datetime expression	Result encoding scheme
Columns	The same encoding scheme as the table that contains the column
Host variables	If the statement references: • A single encoding scheme - The same encoding scheme • Multiple encoding schemes - The application encoding scheme
Special registers	If the statement references: • A single encoding scheme - The same encoding scheme • Multiple encoding schemes - The application encoding scheme
Expressions	If the statement references: • A single encoding scheme - The same encoding scheme • Multiple encoding schemes - The application encoding scheme

For ASCII and EBCDIC, a string representation of a datetime value must be a character string. For Unicode, a string representation of a datetime value can be

either a character string or a graphic string. Thus, the only time a graphic string can be used for a datetime value is when the encoding scheme is Unicode.

In host languages other than Java, a datetime value must be assigned to a string variable. When a date or time is assigned to a string variable, the string format is determined by a precompiler option or subsystem parameter. When a string representation of a datetime value is used in other operations, it is converted to a datetime value. However, this can be done only if the string representation is recognized by DB2 or an exit provided by the installation and the other operand is a compatible datetime value. An input string representation of a date or time with LOCAL specified must have an actual length that is not greater than 255 bytes.

Datetime values that are represented by strings can appear in contexts that require values whose data types are DATE, TIME, TIMESTAMP WITHOUT TIME ZONE, or TIMESTAMP WITH TIME ZONE. A string representation of a date, time or timestamp (with or without time zone) can be passed as an argument to the DATE, TIME, TIMESTAMP, or TIMESTAMP_TZ function to obtain a datetime value. A CAST specification can also be used to turn a character representation of a date, time, or timestamp (with or without time zone) into a datetime value.

Date strings:

A string representation of a date is a string that starts with a digit and has a length of at least 8 characters. Trailing blanks can be included, leading blanks are not allowed, and leading zeros can be omitted in the month and day portions.

The following table shows the valid string formats for dates. Each format is identified by name and includes an associated abbreviation (for use by the CHAR function) and an example of its use. For an installation-defined date string format, the format and length must have been specified when DB2 was installed. They cannot be listed here.

Table 12. Formats for string representations of dates

Format name	Abbreviation	Date format	Example
International Standards Organization	ISO	yyyy-mm-dd	1987-10-12
IBM USA standard	USA	mm/dd/yyyy	10/12/1987
IBM European standard	EUR	dd.mm.yyyy	12.10.1987
Japanese industrial standard Christian era	JIS	yyyy-mm-dd	1987-10-12
Installation-defined	LOCAL	Any installation-defined form	—

Time strings:

A string representation of a time is a string that starts with a digit, and has a length of at least 4 characters. Trailing blanks can be included, leading blanks are not allowed, and leading zeros can be omitted in the hour part of the time; seconds can be omitted entirely. If you choose to omit seconds, an implicit specification of 0 seconds is assumed. Thus 13.30 is equivalent to 13.30.00.

The following table shows the valid string formats for times. Each format is identified by name and includes an associated abbreviation (for use by the CHAR function) and an example of its use. In the case of an

installation-defined time string format, the format and length must have been specified when your DB2 subsystem was installed. They cannot be listed here.

Table 13. Formats for string representations of times

Format name	Abbreviation	Time format	Example
International Standards Organization [1]	ISO[1]	hh.mm.ss	13.30.05
IBM USA standard	USA	hh:mm AM or PM	1:30 PM
IBM European standard	EUR	hh.mm.ss	13.30.05
Japanese industrial standard Christian era	JIS	hh:mm:ss	13:30:05
Installation-defined	LOCAL	Any installation-defined form	—

Note: 1. This is an earlier version of the ISO format. JIS can be used to get the current ISO format.

In the USA format:

- The minutes can be omitted, thereby specifying 00 minutes. For example, 1 PM is equivalent to 1:00 PM.
- The letters A, M, and P can be lowercase.
- A single blank must precede the AM or PM.
- The hour must not be greater than 12 and cannot be 0 except for the special case of 00:00 AM.

Using the ISO format of the 24-hour clock, the correspondence between the USA format and the 24-hour clock is as follows:

- 12:01 AM through 12:59 AM correspond to 00.01.00 through 00.59.00
- 01:00 AM through 11:59 AM correspond to 01.00.00 through 11.59.00
- 12:00 PM (noon) through 11:59 PM correspond to 12.00.00 through 23.59.00
- 12:00 AM (midnight) corresponds to 24.00.00
- 00:00 AM (midnight) corresponds to 00.00.00

Timestamp strings:

A string representation of a timestamp is a character or graphic string that starts with a digit and has a length of at least 16 characters.

The character or graphic string must contain a value that conforms to one of the formats listed in "Datetime constants" on page 151, subject to the following rules:

- leading blanks are not allowed
- trailing blanks can be included
- leading zeros can be omitted from the month, day, hour, and time zone hour elements of the timestamp. An implicit specification of 0 is assumed for any digit that is omitted.
- the hour can be 24 if the minutes, seconds, and any fractional seconds are all zeroes.
- leading zeros must be included for the minute, second, and time zone minute elements of the timestamp.

- the number of digits of fractional seconds can vary from 0 to 12. An implicit specification of 0 is assumed if fractional seconds are omitted.
- the separator character that follows the seconds element can be omitted if fractional seconds are not included.
- an optional single blank can be included between the time and the time zone.
- an optional time zone can be included, in one of the following formats:
  - ±th:tm, with values ranging from -24:00 to +24:00. A value of -0:00 is treated the same as a value of +0:00.
  - ±th, with values ranging from -24 to +24, and an implicit specification of 00 is assumed for the time zone minute element.
  - uppercase Z to specify UTC

If a string representation of a timestamp is implicitly cast to a value with a timestamp data type, the timestamp precision is assumed to be 6, regardless of the number of digits of fractional seconds in the string. Beyond the sixth digit that represents fractional seconds, the digits are truncated and the missing digits are assumed to be zeros. For example, `1990-3-2-8.30.00.10` is equivalent to `1990-03-02-08.30.00.100000`. A string representation of a timestamp can be given a different timestamp precision by explicitly casting the value to a timestamp with a specified precision or, in the case of a constant, preceding the string with the TIMESTAMP keyword (for example, `TIMESTAMP 2007-03-28-14.50.35.123;` has the TIMESTAMP(3) data type).

If a string representation of a timestamp is implicitly cast to a TIMESTAMP WITHOUT TIME ZONE value, the string must not contain a time zone.

SQL statements also support the ODBC or JDBC string representation of a timestamp as an input value only. The ODBC and JDBC string representation of a timestamp has the form *yyyy-mm-dd hh:mm:ss.nnnnnn*.

LOCAL date and time exits: For LOCAL, the date exit for ASCII data is DSNXVDTA, the date exit for EBCDIC is DSNXVDTX, and the date exit for Unicode is DSNXVDTU. For LOCAL, the time exit for ASCII data is DSNXVTMA, the time exit for EBCDIC is DSNXVTMX, and the time exit for Unicode is DSNXVTMU.

Determination of the implicit time zone

DB2 uses the IMPLICIT_TIMEZONE parameter of DSNHDECP to implicitly determines the time zone to associate with a value that does not have a time zone on assignment to a TIMESTAMP WITH TIME ZONE column or variable.

The IMPLICIT_TIMEZONE parameter of DSNHDECP is used to support operations that combine TIMESTAMP WITHOUT TIME ZONE values and TIMESTAMP WITH TIME ZONE values and indicates the time zone to associate with TIMESTAMP WITHOUT TIME ZONE values. For example, on assignment of a value that does not have time zone information (the TIMESTAMP WITHOUT TIME ZONE data type, or a string representation of a timestamp without a time zone) to a TIMESTAMP WITH TIME ZONE target such as a column or variable, DB2 implicitly determines the time zone to associate with the value. The implicit time zone is determined as follows:

- If IMPLICIT_TIMEZONE is not specified or is specified as CURRENT, the implicit time zone is the value of the CURRENT TIME ZONE special register.
- If IMPLICIT_TIMEZONE is specified as SESSION, the implicit time zone is the value of the SESSION TIME ZONE special register.

- If IMPLICIT_TIMEZONE is specified as a character string in the format of '±th:tm' , the implicit time zone is the time zone value represented by the character string.

Restrictions on the use of local datetime formats

Certain restrictions apply on the use of date and time values as input, as output, and for use in binding a package.

The following rules apply to the character-string representation of dates and times:

For input: In distributed operations, DB2 as a server uses its local date or time routine to evaluate host variables and constants. This means that character-string representation of dates and times can be:
- One of the standard formats
- A format recognized by the server's local date/time exit

For output: With DRDA access, DB2 as a server returns date and time host variables in the format defined at the server. To have date and time host variables returned in another format, use `CHAR(date-expression, XXXX)` where *XXXX* is JIS, EUR, USA, ISO, or LOCAL to explicitly specify the specific format.

For BIND PACKAGE COPY: When binding a package using the COPY option, DB2 uses the ISO format for output values unless the SQL statement explicitly specifies a different format. Input values can be specified in the format described previously.

Row ID values

A *row ID* is a value that uniquely identifies a row in a table. A column or a host variable can have a row ID data type.

A ROWID column enables queries to be written that navigate directly to a row in the table because the column implicitly contains the location of the row. Each value in a ROWID column must be unique. Although the location of the row might change, for example across a table space reorganization, DB2 maintains the internal representation of the row ID value permanently. When a row is inserted into the table, DB2 generates a value for the ROWID column unless one is supplied. If a value is supplied, it must be a valid row ID value that was previously generated by DB2 and the column must be defined as GENERATED BY DEFAULT. Users cannot update the value of a ROWID column.

The internal representation of a row ID value is transparent to the user. The value is never subject to character conversion because it is considered to contain BIT data. The length of a ROWID column as described in the LENGTH column of catalog table SYSCOLUMNS is the internal length, which is 17 bytes. The length as described in the LENGTH2 column of catalog table SYSCOLUMNS is the external length, which is 40 bytes.

A ROWID column can be either user-defined or implicitly generated by DB2. You can use the CREATE TABLE statement or the ALTER TABLE statement to define a ROWID column. If you define a LOB column in a table and the table does not have a ROWID column, DB2 implicitly generates a ROWID column. DB2:
- Creates the column with a name of DB2_GENERATED_ROWID_FOR_LOBS*nn*.

  DB2 appends *nn* only if the column name already exists in the table, replacing *nn* with '00' and incrementing by '1' until the name is unique within the row.
- Defines the column as GENERATED ALWAYS.

- Appends the column to the end of the row after all the other explicitly defined columns.

An implicitly hidden ROWID column is called *a hidden ROWID column*. A hidden ROWID column is not visible in SQL statements unless you refer to the column directly by name. For example, assume that DB2 generated a hidden ROWID column named DB2_GENERATED_ROWID_FOR_LOBS for table MYTABLE. The result table for a `SELECT *` statement for table MYTABLE would not contain that ROWID column. However, the result table for `SELECT COL1, DB2_GENERATED_ROWID_FOR_LOBS` would include the hidden ROWID column.

If you add a ROWID column to a table that already has a hidden ROWID column, DB2 ensures that the corresponding values in each column are identical. If the ROWID column that you add is defined as GENERATED BY DEFAULT, DB2 changes the attribute of the hidden ROWID column to GENERATED BY DEFAULT.

For information about using row IDs, see *DB2 Application Programming and SQL Guide*.

XML values

An XML value represents well-formed XML in the form of an XML document, XML content, or a sequence of XML nodes.

An XML value that is stored in a table as the value of a column that is defined with the XML data type must be a well-formed XML document. XML values are processed in an internal representation that is not comparable to any string value. The only predicates that can be applied to the XML data type are the XMLEXISTS predicate and the NULL predicate.

An XML value can be transformed into a serialized string value that represents the XML document by using the XMLSERIALIZE function. Similarly, a string value that represents an XML document can be transformed to an XML value by using the XMLPARSE function.

The XML data type has a variable length and allows for a wide range of sizes. Although data of this type has no defined maximum length, it does have an effective maximum length limit when treated as a serialized string value that represents XML. The maximum effective length is the same as the DB2 limit for a LOB data value. DB2 treats XML string data in a similar manner as LOB data to accommodate very large XML values. Thus, XML values are constrained by the same maximum length limit as LOB data. Unlike the LOB data type which has a LOB locator type, there is no XML locator type.

Restrictions when using XML values: With a few exceptions, you can use XML values in the same contexts in which you can use other data type. XML values cannot be used in the following contexts:

- SELECT lists that are preceded by the DISTINCT clause
- GROUP BY clauses
- ORDER BY clauses
- A subselect of a fullselect with a set operator that is not UNION ALL
- Basic predicates, quantified predicates, BETWEEN predicates, DISTINCT predicates, IN predicates, or LIKE predicates
- Aggregate functions with the DISTINCT keyword

- Primary, unique, or foreign keys
- CREATE TYPE statements

No host languages have any built-in support for an XML data type.

User-defined data types

A *user-defined data type* is a data type that is defined using a CREATE TYPE statement.

The following types of user-defined data type are supported:

- Distinct type
- Array type

Distinct types

A *distinct type* is a user-defined data type that shares its internal representation with a built-in data type (its *source type*), but is considered to be a separate and incompatible data type for most operations.

For example, the semantics for a picture type, a text type, and an audio type that all use the built-in data type BLOB for their internal representation are quite different. A distinct type is created with the SQL statement CREATE TYPE.

For example, the following statement creates a distinct type named AUDIO:

```
CREATE TYPE AUDIO AS BLOB (1M);
```

Although AUDIO has the same representation as the built-in data type BLOB, it is a separate data type that is not comparable to a BLOB or to any other data type. This inability to compare AUDIO to other data types allows functions to be created specifically for AUDIO and assures that these functions cannot be applied to other data types.

The name of a distinct type is qualified with a schema name. The implicit schema name for an unqualified name depends on the context in which the distinct type appears. If an unqualified distinct type name is used:

- In a CREATE TYPE statement or the object of DROP, COMMENT, GRANT, or REVOKE statement, DB2 uses the normal process of qualification by authorization ID to determine the schema name.
- In any other context, DB2 uses the SQL path to determine the schema name. DB2 searches the schemas in the path, in sequence, and selects the first schema in the path such that the distinct type exists in the schema and the user has authorization to use the data type. For a description of the SQL path, see "SQL path" on page 64.

A distinct type does not automatically acquire the functions and operators of its source type because they might not be meaningful. (For example, it might make sense for a "length" function of the AUDIO type to return the length in seconds rather than in bytes.) Instead, distinct types support *strong typing*. Strong typing ensures that only the functions and operators that are explicitly defined on a distinct type can be applied to that distinct type. However, a function or operator of the source type can be applied to the distinct type by creating an appropriate user-defined function. The user-defined function must be sourced on the existing function that has the source type as a parameter. For example, the following series of SQL statements shows how to create a distinct type named MONEY based on

data type DECIMAL(9,2), how to define the + operator for the distinct type, and how the operator might be applied to the distinct type:

```
CREATE TYPE MONEY AS DECIMAL(9,2);
CREATE FUNCTION "+"(MONEY,MONEY)
   RETURNS MONEY
   SOURCE SYSIBM."+"(DECIMAL(9,2),DECIMAL(9,2));
CREATE TABLE SALARY_TABLE
   (SALARY MONEY,
    COMMISSION MONEY);
SELECT SALARY + COMMISSION FROM SALARY_TABLE;
```

A distinct type is subject to the same restrictions as its source type. For example, if a CLOB value is not allowed as input to a function, you cannot specify a distinct type that is based on a CLOB as input.

The comparison operators are automatically generated for distinct types, except those that are based on a CLOB, DBCLOB, or BLOB. In addition, DB2 automatically generates functions for every distinct type that support casting from the source type to the distinct type and from the distinct type to the source type. For example, for the AUDIO type created above, these are generated cast functions:

```
FUNCTION schema-name.BLOB (schema-name.AUDIO) RETURNS SYSIBM.BLOB (1M)
FUNCTION schema-name.AUDIO (SYSIBM.BLOB (1M)) RETURNS schema-name.AUDIO
```

Array types

A user-defined *array type* is a data type that is defined as an array of elements. A user-defined array type can be either an *ordinary array* or *associative array*.

A user-defined ordinary array type has a maximum cardinality, which is specified on the CREATE TYPE (array) statement. A user-defined associative array has a maximum cardinality of 2 billion.

Array values:

An *array value* is a structure that contains an ordered collection of elements.

All elements of an array value must have the same data type. The cardinality of the array is equal to the number of elements in the array.

An array value can be non-empty, empty (cardinality zero), or null. The individual elements in the array can be null or not null. An empty array, an array value of null, and an array for which all elements are the null value are different from each other. An uninitialized array is a null array.

The following example demonstrates the difference between an empty array, a null array, and an array for which individual elements are null.

```
SET PHONELIST = ARRAY[];
           /* Set an entire array to empty           */
SET PHONELIST = NULL;
           /* Set an entire array to the NULL value  */
SET PHONELIST = ARRAY[NULL];
           /* Set one element of an array to NULL    */
SET PHONELIST = ARRAY[NULL, NULL, NULL];
           /* Set three elements of an array to NULL */
```

An *ordinary array* has a defined upper bound on the number of elements, which is known as the maximum cardinality. Each element in the array is referenced by an associated index value that represents the position of that element in the array. The data type of the index values is INTEGER. If *n* is the number of elements in an

ordinary array, the ordinal position that is associated with each element is an integer value greater than or equal to 1 and less than or equal to *n*.

Unlike the maximum cardinality of an array in programming languages such as C, the maximum cardinality of an ordinary array in SQL is not related to the physical representation of the array. The amount of memory that is required to represent the value of an ordinary array is usually proportional to the cardinality of the array, and not to the maximum cardinality of the array type. When an ordinary array is referenced, all of the values in the array are stored in main memory. Therefore, ordinary arrays that contain a large amount of data consume large amounts of main memory.

An *associative array* has no predefined upper bound on the number of elements. An associative array contains an ordered set of zero or more elements, where each element in the array is ordered by and can be referenced by an associated index value. The data type of the index values can be an integer or a character string other than a CLOB, but all index values for the array must have the same data type. The index values of an associative array are unique, and do not need to be contiguous.

A *user-defined array type* is a user-defined data type that is defined as an array. An SQL variable or SQL parameter can be defined as a user-defined array type. Additionally, the result of an invocation of the built-in ARRAY_DELETE or TRIM_ARRAY functions, or the result of a CAST specification, can be a user-defined array type. An element of a user-defined array type can be referenced anywhere that an expression that returns the same data type as an element of that array can be used.

An *unnamed array type* is an array without an associated user-defined data type. The result of invocation of the aggregate built-in function ARRAY_AGG or of an array constructor is an array without an associated user-defined data type. An element of an array without an associated user-defined array type cannot be directly referenced.

The value of an array index can be specified by an expression. That expression can include a reference to a column. If a column is defined with a column mask, the column mask is applied using the normal rules for applying a column mask.

The value of an index for an array element is never null. If an expression specifies a value for an index, and the expression evaluates to the null value, the null value is returned for the array value.

An array value can be specified using one of the following methods:

- A simple reference to an SQL variable, or SQL parameter that is a user-defined array type.
- Invocation of the ARRAY_AGG function.
- Invocation of the ARRAY_DELETE or TRIM_ARRAY built-in functions.
- Use of an array constructor.
- Invocation of a CAST specification that returns an array value.

An array value cannot be stored in a table. An array value can be used only within an SQL PL routine, or when an SQL PL routine is connected to a DRDA server that supports ordinary arrays.

Datetime data in the elements of an array is considered to be CCSID UNICODE (1208).

Related reference:

"Array constructor" on page 280

"ARRAY_AGG" on page 347

"TRIM_ARRAY" on page 658

"CREATE MASK" on page 1299

"CALL" on page 1117

Promotion of data types

Data types can be classified into groups of related data types. Within such groups, an order of precedence exists in which one data type is considered to precede another data type. This precedence enables DB2 to support the *promotion* of one data type to another data type that appears later in the precedence order.

For example, DB2 can promote the data type CHAR to VARCHAR and the data type INTEGER to DOUBLE PRECISION; however, DB2 cannot promote a CLOB to a VARCHAR.

DB2 considers the promotion of data types when:

- Performing function resolution (see "Function resolution" on page 234)
- Casting distinct types (see "Casting between data types" on page 111)
- Assigning built-in data types to distinct types (see "Distinct type assignments" on page 131)

For each data type, the following table shows the precedence list (in order) that DB2 uses to determine the data types to which the data type can be promoted. The table indicates that the best choice is the same data type and not promotion to another data type. The table also shows data types that are considered equivalent during the promotion process. For example, CHARACTER and GRAPHIC are considered to be equivalent data types.

Table 14. Precedence of data types

Data type[1,2]	Data type precedence list (in best-to-worst order)
SMALLINT[3]	SMALLINT, INTEGER, BIGINT, decimal, real, double, DECFLOAT
INTEGER[3]	INTEGER, BIGINT, decimal, real, double, DECFLOAT
BIGINT[3]	BIGINT, decimal, real, double, DECFLOAT
decimal[3]	decimal, real, double, DECFLOAT
real	real, double, DECFLOAT
double	double, DECFLOAT
DECFLOAT	DECFLOAT
CHAR or GRAPHIC	CHAR or GRAPHIC, VARCHAR or VARGRAPHIC, CLOB or DBCLOB
VARCHAR or VARGRAPHIC	VARCHAR or VARGRAPHIC, CLOB or DBCLOB
CLOB or DBCLOB	CLOB or DBCLOB
BINARY	BINARY, VARBINARY, BLOB
VARBINARY	VARBINARY, BLOB

Table 14. Precedence of data types (continued)

Data type[1,2]	Data type precedence list (in best-to-worst order)
BLOB	BLOB
DATE	DATE
TIME	TIME
TIMESTAMP WITHOUT TIME ZONE	TIMESTAMP WITHOUT TIME ZONE or TIMESTAMP WITH TIME ZONE
TIMESTAMP WITH TIME ZONE	TIMESTAMP WITHOUT TIME ZONE or TIMESTAMP WITH TIME ZONE
ROWID	ROWID
XML	XML
A distinct type	The same distinct type

Notes:

1. The data types in lowercase letters represent the following data types:

   decimal DECIMAL(*p*,*s*) or NUMERIC(*p*,*s*)

   real REAL or FLOAT(*n*) where *n* is not greater than 21

   double DOUBLE, DOUBLE PRECISION, FLOAT or FLOAT(*n*) where *n* is greater than 21

2. Other synonyms for the listed data types are considered to be the same as the synonym listed.
3. Real and double are checked for function resolution purposes only. Additionally, the number of significant digits (even for DECFLOAT(16)), and the exponent range of DECFLOAT exceeds that of real and double (double has 16 significant digits). Therefore, DECFLOAT values will not be promoted to real or double.

Casting between data types

There are many occasions when a value with a given data type needs to be *cast* (changed) to a different data type or to the same data type with a different length, precision, or scale.

Data type promotion is one example where the promotion of one data type to another data type requires that the value be cast to the new data type. A data type that can be changed to another data type is *castable* from the source data type to the target data type.

The casting of one data type to another can occur implicitly or explicitly. The cast functions, CAST specification, or XMLCAST specification can be used to explicitly change a data type, depending on the data types involved. In addition, when a sourced user-defined function is created, the data types of the parameters of the source function must be castable to the data types of the function that is being created.

If truncation occurs when a character or graphic string is cast to another data type, a warning occurs if any non-blank characters are truncated. This truncation behavior is similar to retrieval assignment of character or graphic strings. See "Retrieval assignment" on page 128.

If truncation occurs when casting to a binary string, an error is returned.

For casts that involve a distinct type as either the data type to be cast to or from, Table 15 shows the supported casts.

For casting a parameter marker or NULL value to the XML data type, the CAST specification can be used. XML input can also be specified for the CAST specification when the result data type is XML.

Casts that involve an array type as the target and a non-null source value must conform to the following rules:

- If the source value is a user-defined array type:
  - A value of a user-defined array type can only be cast to another user-defined array type of the same type.
  - If the target user-defined array type is an ordinary array, the cardinality of the source array value must be less than or equal to the maximum cardinality of the target array type.
- If the source value is the result of the ARRAY_AGG function or an array_constructor, it is an array without an associated user-defined array type. If the source value is an array without an associated user-defined array type:
  - The target user-defined array type must be an ordinary array. The cardinality of the source array value must be less than or equal to the maximum cardinality of the target array type.
  - The elements in the source array value must be castable to the data type of the elements of the target array type. The index values for the source array value must be castable to the data type of the index of the target array type.

Table 15. Supported casts when a distinct type is involved

Data type ...	**Is castable to data type ...**
Distinct type *DT*	Source data type of distinct type *DT*
Source data type of distinct type *DT*	Distinct type *DT*
Distinct type *DT*	Distinct type *DT*
Data type *A*	Distinct type *DT* where *A* is promotable to the source data type of distinct type *DT* (see "Promotion of data types" on page 110)
INTEGER	Distinct type *DT* if *DT*'s source data type is SMALLINT
DOUBLE	Distinct type *DT* if *DT*'s source data type is REAL
VARCHAR	Distinct type *DT* if *DT*'s source data type is CHAR or GRAPHIC
VARGRAPHIC	Distinct type *DT* if *DT*'s source data type is GRAPHIC or CHAR
VARBINARY	Distinct type *DT* if *DT*'s source data type is BINARY

When a distinct type is involved in a cast, a cast function that was generated when the distinct type was created is used. How DB2 chooses the function depends on whether function notation or CAST specification syntax is used. (For details, see "Function resolution" on page 234 and "CAST specification" on page 267, respectively.) Function resolution is similar for both. However, in CAST specification, when an unqualified distinct type is specified as the target data type, DB2 first resolves the schema name of the distinct type and then uses that schema name to locate the cast function.

For casts between built-in data types, the following table shows the supported casts.

Table 16. Supported casts between built-in data types

	To data type[1]																					
Cast from data type –	**SMALLINT**	**INTEGER**	**BIGINT**	**DECIMAL**	**DECFLOAT**	**REAL**	**DOUBLE**	**CHAR**	**VARCHAR**	**CLOB**	**GRAPHIC**	**VARGRAPHIC**	**DBCLOB**	**BINARY**	**VARBINARY**	**BLOB**	**DATE**	**TIME**	**TIMESTAMP WITHOUT TIME ZONE**	**TIMESTAMP WITH TIME ZONE**	**ROWID**	**XML**
SMALLINT	Y	Y	Y	Y	Y	Y	Y	Y	Y													
INTEGER	Y	Y	Y	Y	Y	Y	Y	Y	Y													
BIGINT	Y	Y	Y	Y	Y	Y	Y	Y	Y													
DECIMAL	Y	Y	Y	Y	Y	Y	Y	Y	Y													
DECFLOAT	Y	Y	Y	Y	Y	Y	Y	Y	Y													
REAL	Y	Y	Y	Y	Y	Y	Y	Y	Y													
DOUBLE	Y	Y	Y	Y	Y	Y	Y	Y	Y													
CHAR	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	
VARCHAR	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	
CLOB								Y	Y	Y	Y	Y	Y	Y	Y	Y						
GRAPHIC	Y	Y	Y	Y	Y	Y	Y	Y[2]	Y[2]	Y[2]	Y	Y	Y	Y	Y	Y	Y[3]	Y[3]	Y[3]	Y[3]		
VARGRAPHIC	Y	Y	Y	Y	Y	Y	Y	Y[2]	Y[2]	Y[2]	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y[3]		
DBCLOB								Y[2]	Y[2]	Y[2]	Y	Y	Y	Y	Y	Y						
BINARY														Y	Y	Y						
VARBINARY														Y	Y	Y						
BLOB														Y	Y	Y						
DATE								Y	Y								Y		Y			
TIME								Y	Y									Y				
TIMESTAMP WITHOUT TIME ZONE								Y	Y								Y	Y	Y	Y		
TIMESTAMP WITH TIME ZONE																						

Table 16. Supported casts between built-in data types (continued)

Cast from data type –	To data type[1] SMALLINT	INTEGER	BIGINT	DECIMAL	DECFLOAT	REAL	DOUBLE	CHAR	VARCHAR	CLOB	GRAPHIC	VARGRAPHIC	DBCLOB	BINARY	VARBINARY	BLOB	DATE	TIME	TIMESTAMP WITHOUT TIME ZONE	TIMESTAMP WITH TIME ZONE	ROWID	XML
TIMESTAMP WITH TIME ZONE								Y	Y								Y	Y	Y	Y		
ROWID								Y	Y					Y	Y	Y					Y	
XML																						Y

Note:

1. Other synonyms for the listed data types are considered to be the same as the synonym listed. Some exceptions exist when the cast involves character string data if the subtype is FOR BIT DATA.
2. The result length for these casts is `3 * LENGTH(graphic string)`.
3. These data types are castable between each other only if the data is Unicode.

Table 17 shows where to find information about the rules that apply when casting to the identified target data types.

Table 17. Rules for casting to a data type

Target data type	Rules
SMALLINT	"SMALLINT" on page 608
INTEGER	"INTEGER or INT" on page 496
BIGINT	"BIGINT" on page 387
DECIMAL	"DECIMAL or DEC" on page 447
NUMERIC	"DECIMAL or DEC" on page 447
REAL	"REAL" on page 580

Table 17. Rules for casting to a data type (continued)

Target data type	Rules
DOUBLE	"DOUBLE_PRECISION or DOUBLE" on page 458
DECFLOAT	"DECFLOAT" on page 440
CHAR	"CHAR" on page 398
VARCHAR	"VARCHAR" on page 673
CLOB	"CLOB" on page 409
GRAPHIC	"GRAPHIC" on page 479
VARGRAPHIC	"VARGRAPHIC" on page 690
DBCLOB	"DBCLOB" on page 436
BINARY	"BINARY" on page 389
VARBINARY	"VARBINARY" on page 671
BLOB	"BLOB" on page 393
DATE	"DATE" on page 425
TIME	"TIME" on page 629
TIMESTAMP WITHOUT TIME ZONE	If the source data type is a character or graphic string, see "TIMESTAMP" on page 630, where one operand is specified. If the string contains a time zone, an error is returned. If the source data type is a DATE, the timestamp is composed of the specified date and a time of 00:00:00. If the source is a TIMESTAMP WITH TIME ZONE, the resulting timestamp is the timestamp without time zone element of the specified datetime value, which is the local timestamp in the corresponding time zone. For example: `cast('2008-04-12-07.30.00.0-6:00' as TIMESTAMP)` returns `2008-04-12-07.30.00.0`. If the source type is a TIMESTAMP WITHOUT TIME ZONE the timestamp is the specified value.
TIMESTAMP WITH TIME ZONE	If the source data type is a character or graphic string or TIMESTAMP WITHOUT TIME ZONE, see "TIMESTAMP_TZ" on page 645, where one operand is specified. If the string contains a time zone, an error is returned. If the source type is a TIMESTAMP WITH TIME ZONE, the timestamp is the specified value.
ROWID	"ROWID" on page 595

Table 18. The derived length of an argument when a built-in scalar function is invoked and implicit casting is required.

	Target data type								
Source data type	**CHAR**	**GRAPHIC**	**VARCHAR**	**VAR-GRAPHIC**	**CLOB**	**DBCLOB**	**BLOB**	**TIME STAMP (*precision*)**	**DECFLOAT**
SMALLINT	6	6	6	6					
INTEGER	11	11	11	11					
BIGINT	20	20	20	20					
DECIMAL (*p*,*s*)	2+*p*	2+*p*	2+*p*	2+*p*					
REAL	24	24	24	24					
DOUBLE	24	24	24	24					
DECFLOAT	42	42	42	42					

Table 18. The derived length of an argument when a built-in scalar function is invoked and implicit casting is required. (continued)

	Target data type								
Source data type	**CHAR**	**GRAPHIC**	**VARCHAR**	**VAR-GRAPHIC**	**CLOB**	**DBCLOB**	**BLOB**	**TIME STAMP (*precision*)**	**DECFLOAT**
CHAR(*n*)								12	34
VARCHAR (*n*)	min(*n*,254)							12	34
CLOB(*n*)									
GRAPHIC (*n*)								12	34
VARGRAPHIC (*n*)								12	34
DBCLOB (*n*)									
BLOB(*n*)									
TIME	8	8	8	8					
DATE	10	10	10	10					
TIME-STAMP(*p*) WITHOUT TIME ZONE	If *p*=0 then 19, otherwise 20+*p*	If *p*=0 then 19, otherwise 20+*p*	If *p*=0 then 19, otherwise 20+*p*	If *p*=0 then 19, otherwise 20+*p*					
TIME-STAMP(*p*) WITH TIME ZONE	If *p*=0 then 148, otherwise 149+*p*	If *p*=0 then 148, otherwise 149+*p*	If *p*=0 then 148, otherwise 149+*p*	If *p*=0 then 148, otherwise 149+*p*					

Casting non-XML values to XML values

Table 19. Supported Casts from Non-XML Values to XML Values

	Target Data Type	
Source Data Type	**XML**	**Resulting XML Schema Type**
SMALLINT	Y	xs:short
INTEGER	Y	xs:int
BIGINT	Y	xs:long
DECIMAL	Y	xs:decimal
DECFLOAT	N	
REAL	N	
FLOAT	Y	xs:double
DOUBLE	Y	xs:double
CHAR	Y	xs:string
VARCHAR	Y	xs:string
CLOB	Y	xs:string
GRAPHIC	Y	xs:string
VARGRAPHIC	Y	xs:string
DBCLOB	Y	xs:string
BINARY	N	
VARBINARY	N	
BLOB	N	

Table 19. Supported Casts from Non-XML Values to XML Values (continued)

	Target Data Type	
Source Data Type	**XML**	**Resulting XML Schema Type**
character type FOR BIT DATA	N	
DATE	N	
TIME	N	
TIMESTAMP WITHOUT TIME ZONE	N	
TIMESTAMP WITH TIME ZONE	N	
ROWID	N	
distinct type	N	

When character string values are cast to XML values, the resulting xs:string atomic value cannot contain illegal XML characters. If the input character string is not in Unicode, the input characters are converted to Unicode.

Casting XML values to non-XML values

An XMLCAST from an XML value to a non-XML value can be described as two casts: an XQuery cast that converts the source XML value to a target XQuery data type that corresponds to the SQL target type, followed by a cast from the corresponding XQuery data type to the actual SQL type. The target XQuery data type is an XML schema data type like xs:decimal or xs:string, as shown in the follow table.

An XMLCAST is supported if the target type has a corresponding XQuery target type that is supported, and if there is a supported XQuery cast from the type of the source value to the corresponding XQuery target type. The target type that is used in the XQuery cast is based on the corresponding XQuery target type and might contain some additional restrictions.

The following table lists the XQuery types that result from such conversion.

Table 20. Supported Casts from XML Values to Non-XML Values

	Source Data Type	
Target Data Type	**XML**	**Corresponding XQuery Target Type**
SMALLINT	Y	xs:integer
INTEGER	Y	xs:integer
BIGINT	Y	xs:integer
DECIMAL	Y	xs:decimal
DECFLOAT	Y	xs:double
REAL	Y	xs:double
FLOAT	Y	xs:double
DOUBLE	Y	xs:double
CHAR	Y	xs:string
VARCHAR	Y	xs:string
CLOB	Y	xs:string
GRAPHIC	Y	xs:string

Table 20. Supported Casts from XML Values to Non-XML Values (continued)

	Source Data Type	
Target Data Type	**XML**	**Corresponding XQuery Target Type**
VARGRAPHIC	Y	xs:string
DBCLOB	Y	xs:string
BINARY	N	
VARBINARY	N	
BLOB	N	
character type FOR BIT DATA	N	
DATE	Y	xs:date
TIME	Y	xs:time
TIMESTAMP	Y	xs:dateTime
ROWID	N	
distinct type	N	

The following restrictions are in effect when a value is cast from an XQuery target data type to a target SQL data type:

- If the target type is one of the character or graphic string types, the resulting XML value is converted, if necessary, to the CCSID of the target data type using the rules described in "Conversion rules for string assignment" on page 129, before it is converted to the target type with a limited length. Truncation occurs if the specified length limit is smaller than the length of the resulting string after CCSID conversion. A warning occurs if any non-blank characters are truncated. If the target type is a fixed-length string type (CHAR or GRAPHIC) and the specified length of the target type is greater than the length of the resulting string from CCSID conversion, blanks are padded at the end. This truncation and padding behavior is similar to retrieval assignment of character or graphic strings.
- If the target type is DOUBLE or REAL and the source XML value after the XQuery cast is an xs:double value of INF, -INF, or NaN, an error is returned. If the source value is an xs:double negative zero, the value is converted to positive zero. If the source value is beyond the range of the target data type, an overflow error is returned. If the source value contains more significant digits than the precision of the target data type, the source value is rounded to the precision of the target data type.
- If the target type is DECFLOAT and the source XML value is an xs:double value of INF, -INF, or NaN, the result will be the corresponding special DECFLOAT values INF, -INF, or NaN. If the source value is an xs:double negative zero, the result is negative zero. If the target type is DECFLOAT(16) and the source value is beyond the range of DECFLOAT(16), an overflow error is returned. If the source value has more than 16 significant digits, the value is rounded according to the ROUNDING mode that is in effect. This rounding behavior is the same as what is used during the cast of DECFLOAT(34) to DECFLOAT(16).
- If the target type is DECIMAL, the resulting xs:decimal value is converted, if necessary, to the precision and scale of the target data type. The necessary number of leading zeros is added or removed. In the fractional part of the number, the necessary number of trailing zeros is added or the necessary number of digits is eliminated. This truncation behavior is similar to the behavior of the cast from DECIMAL to DECIMAL.

- If the target type is DATE, TIME, or TIMESTAMP WITHOUT TIME ZONE, the resulting XML value is adjusted to UTC time and the time zone component is removed. If the source does not include a time zone and the target data type is TIMESTAMP WITH TIME ZONE, zeroes are used for the time zone component. If the target type is TIME and the resulting XML value contains a seconds component with non-zero digits after the decimal point, those digits are truncated. If the target type is DATE or timestamp, the year part of the resulting xs:date or xs:dateTime value must be in the range of 0001 to 9999. If the target type is timestamp and the precision of the target timestamp is less than 12, the fractional seconds part of the xs:dateTime value will be truncated to the target timestamp precision.

Implicit cast from numeric data to string data

When DB2 implicitly casts a numeric value to a string value, the target type is VARCHAR value which is then compatible with other character string or graphic string data types.

The length attribute and the CCSID attribute of the result of the cast are determined in the same way as the VARCHAR function. When GRAPHIC or VARGRAPHIC data types are involved, the encoding scheme must be UNICODE. The following table shows the target type and length:

Table 21. Target type and length attribute for implicit cast from numeric types to string types

Source data type	Target data type
SMALLINT	VARCHAR(6)
INTEGER	VARCHAR(11)
BIGINT	VARCHAR(20)
NUMERIC or DECIMAL	VARCHAR(*precision*+2)
REAL	VARCHAR(24)
FLOAT	VARCHAR(24)
DOUBLE	VARCHAR(24)
DECFLOAT	VARCHAR(42)

Implicit conversion from a numeric value to a string value can happen during:

- Assignment (where the source value is a number and the target operand is a character string or graphic string data type).

  Among assignment statements, implicit casting is not supported for the SET statements for special registers, the RETURNS clause and RETURN statement for functions, and the SQL control statements: RETURN, SIGNAL, and RESIGNAL.
- Application of concatenation operators (CONCAT and ||)
- Application of set operators.

  Implicit conversion from a numeric value to a string value is not supported for an operand of a set operator.

Implicit cast from string data to numeric data

When DB2 implicitly casts a character string or graphic string value to a numeric value, the target type is DECFLOAT(34) which is compatible with other numeric data types.

When GRAPHIC or VARGRAPHIC data types are involved, the encoding scheme must be UNICODE. The following table shows the target type and length:

Table 22. Target type and length attribute for implicit cast from string types to numeric types

Source data type	Target data type
CHAR	DECFLOAT(34)
VARCHAR	DECFLOAT(34)
GRAPHIC	DECFLOAT(34)
VARGRAPHIC	DECFLOAT(34)
CHAR FOR BIT DATA or VARCHAR FOR BIT DATA	N/A
BINARY	N/A
VARBINARY	N/A
BLOB	N/A
CLOB	N/A
DBCLOB	N/A

Implicit conversion from a string value to a numeric value can happen during:

- Assignment (where the source value is a character string or graphic string and the target operand is a numeric data type).
- Comparisons

  When a character string or graphic string value is compared with a numeric value, DB2 implicitly converts the string value to DECFLOAT(34) and applies numeric comparison rule between the DECFLOAT(34) value and the other numeric value.

  – Basic predicates, quantified predicates, and DISTINCT predicates (one operand is numeric value and the other operand is character string or graphic string value)

    Numeric is the dominant data type. The character or graphic string value is cast to DECFLOAT(34) value.

  – BETWEEN predicates

    Numeric is the dominant data type. If any of the three operands is a numeric value, DB2 implicitly casts the character or graphic string operands to DECFLOAT data type.

  – IN predicates

    Numeric is the dominant data type. If any of the operands is a numeric value, DB2 implicitly casts the character or graphic string operands to DECFLOAT data type.

  – *Searched-when-clause* of CASE expression

    Pair-wise comparison is performed. Implicit cast of each pair follows the same rule as for a basic predicate. Implicit string and numeric cast is supported on *searched-when-clause* of CASE expression.

  – Search conditions in SQL control statements (one operand is numeric value and the other operand is character string or graphic string value)

    The search condition can appear in SQL control statements like the CASE statement, the IF statement, the REPEAT statement, and the WHILE statement. For comparisons in the search condition, numeric is the dominant data type. The character string or graphic string value is cast to a

DECFLOAT(34) value. Implicit string and numeric cast is supported on the *searched-when-clause* of the CASE statement.

- Arithmetic operators (unary arithmetic operators + and - and infix arithmetic operators +, -, *, and /)

  If the operand of unary arithmetic operators is of a character string or graphic string data type, that operand is implicitly cast to DECFLOAT(34). For infix arithmetic operators, if one operand is a numeric value or both operands are character or graphic string values, DB2 implicitly casts the character string or graphic string operand to the DECFLOAT data type.

Implicit conversion from a string value to a numeric value is not supported for an operand of a set operator.

Assignment and comparison

The basic operations of SQL are assignment and comparison.

Assignment operations are performed during the execution of statements such as CALL, INSERT, UPDATE, MERGE, FETCH, SELECT INTO, SET *host-variable* or SET *assignment-statement*, and VALUES INTO statements. In addition, when a function is invoked or a stored procedure is called, the arguments of the function or stored procedure are assigned. Comparison operations are performed during the execution of statements that include predicates and other language elements such as MAX, MIN, DISTINCT, GROUP BY, and ORDER BY.

The basic rule for both operations is that data types of the operands must be compatible. The compatibility rule also applies to other operations that are described under "Rules for result data types" on page 144.

The following table shows the compatibility of data types for assignments and comparisons.

Table 23. Data Type Compatibility for Assignments and Comparisons

Operand	Binary integer	Decimal number	Floating point	Decimal floating point	Character string	Graphic string	Binary string	Date	Time	Timestamp without time zone	Timestamp with time zone	Row ID	User-defined type	XML[8]
Binary integer	Yes	Yes	Yes	Yes	[1]	[1]	No	No	No	No	No	No	[2]	No
Decimal number	Yes	Yes	Yes	Yes	[1]	[1]	No	No	No	No	No	No	[2]	No
Floating point	Yes	Yes	Yes	Yes	[1]	[1]	No	No	No	No	No	No	[2]	No
Decimal floating point	Yes	Yes	Yes	Yes	[1]	[1]	No	No	No	No	No	No	[2]	No
Character string	[1]	[1]	[1]	[1]	Yes	Yes [3,4]	No [5]	Yes	Yes	Yes	Yes	No	[2]	No
Graphic string	[6]	[6]	[6]	[6]	Yes [3,4]	Yes	No	[3]	[3]	[3]	[3]	No	[2]	No
Binary string	No	No	No	No	No [5]	No	Yes	No	No	No	No	No	[2]	No
Date	No	No	No	No	[7]	[3,7]	No	Yes	No	No	No	No	[2]	No
Time	No	No	No	No	[7]	[3,7]	No	No	Yes	No	No	No	[2]	No
Timestamp without time zone	No	No	No	No	[7]	[3,7]	No	No	No	Yes	Yes	No	[2]	No
Timestamp with time zone	No	No	No	No	[7]	[3,7]	No	No	No	Yes	Yes	No	[2]	No
Row ID	No	No	No	No	No	No	No	No	No	No	No	Yes	[2]	No
User-defined type	[2]	[2]	[2]	[2]	[2]	[2]	[2]	[2]	[2]	[2]	[2]	[2]	Yes[2]	No

Table 23. Data Type Compatibility for Assignments and Comparisons (continued)

Operand	Binary integer	Decimal number	Floating point	Decimal floating point	Character string	Graphic string	Binary string	Date	Time	Timestamp without time zone	Timestamp with time zone	Row ID	User-defined type	XML[8]
XML[8]	No	No	No	No	No	No	No	No	No	No	No	No	No	Yes

1. LOBs and bit data are not supported.
2. The compatibility rules for user-defined types are as follows:
   - A user-defined distinct type value is only comparable to a value that is defined with the same user-defined distinct type. In general, assignments are supported between a distinct type value and its source data type.
   - A user-defined array type value is only comparable to a value that is defined with the same user-defined array type.

     This means that in general, an ordinary array type is not compatible with an associative array type. The following exceptions apply only to a CALL statement for a remote stored procedure:

     – A source value that is an ordinary array with an integer index can be specified for a target that is defined as an associative array, if the definitions of the array elements of the two arrays are compatible. DB2 transforms the ordinary array into an associative array with an integer index, preserving the ordering of the elements in the original ordinary array. However, if the associative array is defined with a VARCHAR index, an SQL error code is returned.

     – A source value that is an associative array can be specified for a target defined as an ordinary array . DB2 transforms the associative array into an ordinary array, by assigning the values of the array elements in the associative array in the same order in the target ordinary array, and assigning appropriate index values.

   For additional information, see "User-defined type assignments" on page 131.
3. On assignment and comparison from Graphic to Character, the resulting length in bytes is `3 * (LENGTH(graphic-string))`, depending on the CCSIDs.
4. Character strings with subtype FOR BIT DATA are not compatible with Graphic Data.
5. All character strings, even those with subtype FOR BIT DATA, are not compatible with binary strings.
6. LOBs are not supported.
7. The compatibility of datetime values is limited to assignment and comparison:
   - Datetime values can be assigned to string columns and to string variables that are not LOB values, as explained in "Datetime assignments" on page 129.
   - A valid string representation of a date can be assigned to a date column or compared to a date.
   - A valid string representation of a time can be assigned to a time column or compared to a time.
   - A valid string representation of a timestamp (without or with time zone) can be assigned to a timestamp column (without or with time zone) or compared to a timestamp (without or with time zone).
8. Character and graphic strings, including LOBs, can be assigned to XML columns. For comparison, XML can only be compared using the XMLEXISTS and NULL predicates.

Compatibility with a column that has a field procedure is determined by the data type of the column, which applies to the decoded form of its values.

A basic rule for assignment operations is that a null value cannot be assigned to:

- A column that cannot contain null values
- A non-Java host variable that does not have an associated indicator variable

  For a host variable that does have an associated indicator variable, a null value is assigned by setting the indicator variable to a negative value. See "References to host variables" on page 215 for a discussion of indicator variables.
- A Java host variable that is a primitive type

  For a Java host variable that is not a primitive type, the value of that variable is set to a Java null value.

Numeric assignments

The basic rule for numeric assignments is that the whole part of a decimal or integer number cannot be truncated. If necessary, the fractional part of a decimal number is truncated.

Decimal or integer to floating-point

Because floating-point numbers are only approximations of real numbers, the result of assigning a decimal or integer number to a floating-point column or variable might not be identical to the original number.

Floating-point or decimal to integer

When a single precision floating-point number is converted to integer, rounding occurs on the seventh significant digit, zeros are added to the end of the number, if necessary, starting from the seventh significant digit, and the fractional part of the number is eliminated. When a double precision floating-point or decimal number is converted to integer, the fractional part of the number is eliminated.

Example 1: The following example shows single precision floating-point numbers converted to an integer:

```
Floating-point number:          Results when assigned to an integer column
                                 or host variable:
2.0000045E6                     2000000
2.00000555E8                    200001000
```

Example 2: The following example shows a double precision floating-point number converted to an integer:

```
Floating-point number:          Results when assigned to an integer column
                                 or host variable:
2.0000045E6                     2000004
2.00000555E8                    200000555
```

Example 3: The following example shows a decimal number converted to an integer:

```
Decimal number:                 Results when assigned to an integer column
                                 or host variable:
2000004.5                       2000004
200000555.0                     200000555
```

Decimal to decimal

When a decimal number is assigned to a decimal column or variable, the number is converted, if necessary, to the precision and the scale of the target.

The necessary number of leading zeros is added or eliminated, and, in the fractional part of the number, the necessary number of trailing zeros is added, or the necessary number of trailing digits is eliminated.

Decimal to DECFLOAT

When a decimal number is assigned to a DECFLOAT column or variable, the number is converted to the precision (16 or 34) of the target. Leading zeros are eliminated.

Depending on the precision and scale of the decimal number, and the precision of the target, the value might be rounded to fit.

For static SQL statements other than CREATE VIEW, the ROUNDING bind option or the native SQL procedure option determines the rounding mode.

For dynamic SQL statements (and static CREATE VIEW statements), the special register CURRENT DECFLOAT ROUNDING MODE determines the rounding mode.

Integer to decimal

When an integer is assigned to a decimal column or variable, the number is converted first to a temporary decimal number and then, if necessary, to the precision and scale of the target.

The precision and scale of the temporary decimal number is 5,0 for a small integer, 11,0 for a large integer, or 19,0 for a big integer.

Integer to DECFLOAT

When an integer is assigned to a DECFLOAT column or variable, the number is converted first to a temporary decimal number and then to DECFLOAT.

The precision and scale of the temporary decimal number is 5,0 for a small integer, 11,0 for a large integer, or 19,0 for a big integer. The decimal number is then converted to DECFLOAT using the rules for "Decimal to DECFLOAT" on page 123.

Floating-point to floating-point

When a single precision floating-point number is assigned to a double precision floating-point column or variable, the single precision data is padded with eight hex zeros. When a double precision floating-point number is assigned to a single precision floating-point column or variable, the double precision data is converted and rounded up on the seventh hex digit.

In assembler, C, or C++ applications that are prepared with the FLOAT(IEEE) SQL processing option, floating-point constants and values in host variables are assumed to have IEEE floating-point format. All floating-point data is stored in DB2 in System/390 floating-point format. Therefore, when the FLOAT(IEEE) SQL processing option is in effect, DB2 performs the following conversions:

- When a number in short or long IEEE floating-point format is assigned to a single-precision or double-precision floating-point column, DB2 converts the number to System/390 floating-point format.
- When a single-precision or double-precision floating-point column value is assigned to a short or long floating-point host variable, DB2 converts the column value to IEEE floating-point format.

Floating-point to decimal

When a single precision floating-point number is assigned to a decimal column or variable, the number is first converted to a temporary decimal number.

When a single precision floating-point number is assigned to a decimal column or variable, the number is first converted to a temporary decimal number of precision 6 by rounding on the seventh decimal digit. Twenty five zeros are then appended to the number to bring the precision to 31. Because of rounding, a number less than 0.5×10^{-6} is reduced to 0.

When a double precision floating-point number is assigned to a decimal column or variable, the number is first converted to a temporary decimal number of precision 15, and then, if necessary, truncated to the precision and scale of the target. In this conversion, zeros are added to the end of the number, if necessary, to bring the precision to 16. The number is then rounded (using floating-point arithmetic) on the sixteenth decimal digit to produce a 15-digit number. Because of rounding, a number less in magnitude than 0.5×10^{-15} is reduced to 0. If the decimal number requires more than 15 digits to the left of the decimal point, an error is reported. Otherwise, the scale is given the largest possible value that allows the whole part of the number to be represented without loss of significance.

The following examples show the effect of converting a double precision floating-point number to decimal:

Example 1: The floating-point number, `.123456789098765E-05` in decimal notation is, `.00000123456789098765`. Rounding adds 5 in the 16th position, so the number becomes `.00000123456789148765` and truncates the result to `.000001234567891`. Zeros are then added to the end of a 31-digit result, and the number becomes `.0000012345678910000000000000000`.

Example 2: The floating-point number, `1.2339999999999E+01` in decimal notation is, `12.33999999999900`. Rounding adds 5 in the 16th position, so the number becomes

12.33999999999905 and truncates the result to 12.3399999999990. Zeros are then added to the end of a 31-digit result and the number becomes 12.3399999999990000000000000000.

Floating point to DECFLOAT

When a single or double precision floating-point number is assigned to a DECFLOAT column or variable, the number is first converted to a temporary string representation of the floating point number. The string representation of the number is then converted to DECFLOAT.

DECFLOAT to integer

When a DECFLOAT is assigned to a binary integer column or variable, the fractional part of the number is lost.

Example 1: The following example shows decimal floating-point numbers converted to an integer:

```
Decimal floating-point number:          Results when assigned to an integer column
                                        or host variable:
2.0000045E6                             2000004
2.00000555E8                            200000555
```

DECFLOAT to decimal

When a DECFLOAT value is assigned to a decimal column or variable, the DECFLOAT value is converted, if necessary, to the precision and the scale of the target.

During the assignment, the necessary number of leading zeros is added and, in the fractional part of the number, the necessary number of trailing zeros is added, or rounding occurs.

For static SQL statements other than CREATE VIEW, the ROUNDING bind option or the native SQL procedure option determines the rounding mode.

For dynamic SQL statements (and static CREATE VIEW statements), the special register CURRENT DECFLOAT ROUNDING MODE determines the rounding mode.

Example 1: The following example shows decimal floating-point numbers converted to a decimal value:

```
Decimal floating-point number:          Results when assigned to an decimal(15,0)
                                        column or host variable:
2.0000045E6                             2000005
Decimal floating-point number:          Results when assigned to an decimal(15,2)
                                        column or host variable:
2.0000045E6                             2000004.50
2.00000555E8                            200000555.00
```

DECFLOAT to floating-point

Because floating-point numbers are only approximations of real numbers, the result of assigning a DECFLOAT value to a floating-point column or variable might not be identical to the original number.

The DECFLOAT value is first converted to a string representation, and is then converted to floating-point number.

DECFLOAT(16) to DECFLOAT(34)

When a DECFLOAT(16) is assigned to a DECFLOAT(34) column or variable, the exponent of the source is converted to the corresponding exponent in the result format, and the coefficient is extended by appending zeros on the left.

DECFLOAT(34) to DECFLOAT(16)

When a DECFLOAT(34) is assigned to a DECFLOAT(16) column or variable, the exponent of the source is converted to the corresponding exponent in the result format.

The source coefficient is rounded to the precision of the target.

For static SQL statements, the ROUNDING bind option or the native SQL procedure option determines the rounding mode.

For static SQL statements other than CREATE VIEW, the ROUNDING bind option or the native SQL procedure option determines the rounding mode.

For dynamic SQL statements (and static CREATE VIEW statements), the special register CURRENT DECFLOAT ROUNDING MODE determines the rounding mode.

To COBOL integers

Assignment to COBOL integer variables uses the full size of the integer.

Thus, the value placed in the COBOL data item might be out of the range of values.

COBOL supports some data types with no SQL equivalent (BINARY decimal and DISPLAY decimal data items, for example). In most cases, you can use COBOL statements to convert between the unsupported COBOL data types and the data types that SQL supports.

For DB2 for z/OS, the only BINARY numeric variables allowed as HOST variable are integer binary variables. The only DECIMAL host variables supported by SQL are packed decimal host variables.

Example 1: If COL1 contains a value of 12345, the following statements cause the value 12345 to be placed in A, even though A has been defined with only 4 digits:

```
01  A  PIC  S9999  BINARY.
EXEC SQL SELECT COL1
         INTO :A
         FROM TABLEX
END-EXEC.
```

Example 2: The following COBOL statement results in 2345 being placed in A:

```
MOVE 12345 TO A.
```

String assignments

There are two types of string assignments; storage assignment and retrieval assignment.

- *Storage assignment* is when a value is assigned to a column or a parameter of a function or stored procedure.
- *Retrieval assignment* is when a value is assigned to a variable.

The rules differ for storage and retrieval assignment.

Binary string assignment

Binary string assignment involves assignment at both the storage and the retrieval of binary strings.

Storage assignment:

The length of a string that is assigned to a column or parameter of a function or procedure must not be greater than the length attribute of the column or parameter. If the string is longer than the length attribute of that column or parameter, an error is returned.

When the string is assigned to a fixed-length binary string column or parameter of a function or procedure, and the length of the string is less than the length attribute of that column or parameter, the string is padded to the right with the necessary number of binary zeros.

Retrieval assignment:

The length of a string that is assigned to a variable can be greater than the length attribute of the variable. When a string is assigned to a variable and the string is longer than the length attribute of the variable, the string is truncated on the right by the necessary number of bytes. When this occurs, a warning is returned.

Character and graphic string assignment

The rules for storage and retrieval assignment apply when both the source and the target are strings.

When a datetime data type is involved, see "Datetime assignments" on page 129. For the special considerations that apply when a distinct type is involved in an assignment, especially to a variable, see "Distinct type assignments" on page 131.

Storage assignment:

The basic rule for character storage assignment is that the length of a string that is assigned to a column or parameter of a function or stored procedure must not be greater than the length attribute of the column or the parameter.

Trailing blanks are included in the length of the string. When the length of the string is greater than the length attribute of the column or the parameter, the following actions occur:

- If all of the trailing characters that must be truncated to make a string fit the target are blanks and the string is a character or graphic string, the string is truncated and assigned without warning.
- Otherwise, the string is not assigned and an error occurs to indicate that at least one of the excess characters is non-blank.

When a string is assigned to a fixed-length column or parameter and the length of the string is less than the length attribute of the target, the string is padded to the right with the necessary number of SBCS or DBCS blanks. The pad character is always a blank even for columns or parameters that are defined with the FOR BIT DATA attribute.

Retrieval assignment:

The length of a string that is assigned to a host variable can be greater than the length attribute of the variable. When the length of the string is greater than the length of the variable, the string is truncated on the right by the necessary number of SBCS or DBCS characters.

When truncation occurs, the value W is assigned to the SQLWARN1 field of the SQLCA. Furthermore, if an indicator variable is provided and the source of the value is not a LOB, the indicator variable is set to the original length of the string. The truncation result of an improperly formed mixed string is unpredictable.

When a character string is assigned to a fixed-length variable and the length of the string is less than the length attribute of the target, the string is padded to the right with the necessary number of blanks. The pad character is always a blank even for strings defined with the FOR BIT DATA attribute.

When a string of length *n* is assigned to a varying-length string variable with a maximum length greater than *n*, the characters after the *n*th character of the variable are undefined.

Assignments involving mixed data strings

A mixed data string that contains MBCS characters cannot be assigned to an SBCS column, SBCS parameter, or SBCS variable.

The following rules apply when a mixed data string is assigned to a variable and the string is longer than the length attribute of the variable:

- If the string is not well-formed mixed data, it is truncated as if it were BIT or graphic data.
- If the string is well-formed mixed data, it is truncated on the right such that it is well-formed mixed data with a length that is the same as the length attribute of the variable and the number of characters lost is minimal.

Assignments involving C NUL-terminated strings

A C NUL-terminated string variable that is referenced in a CONNECT statement does not need to contain a NUL. Otherwise, DB2 enforces the convention that the value of a NUL-terminated string variable, either character or graphic, is NUL-terminated.

An input variable that does not contain a NUL will cause an error. A value that is assigned to an output variable will always be NUL-terminated even if a character must be truncated to make room for the NUL.

When a string of length *n* is assigned to a C NUL-terminated string variable with a length greater than *n*+1, the rules depend on whether the source string is a value of a fixed-length string or a varying-length string:

- If the source is a fixed-length string and the value of field PAD NUL-TERMINATED on installation panel DSNTIP4 is YES, the string is padded on the right with *x*-*n*-1 blanks, where *x* is the length of the variable. The padded string is then assigned to the variable and a NUL is appended at the end of the variable. If the value of field PAD NUL-TERMINATED is NO, the string is assigned to the first *n* bytes of the variable and a NUL is appended at the end of the variable.
- If the source is a varying-length string, the string is assigned to the first *n* bytes of the variable and a NUL is appended at the end of the variable.

Conversion rules for string assignment

A character or graphic string that is assigned to a column or variable is first converted, if necessary, to the coded character set of the target. Conversion is necessary only if certain conditions apply.

Conversion is necessary only if all the following are true:
- The CCSIDs of string and target are different.
- Neither CCSID is X'FFFF' (neither the string nor the target is defined as BIT data).
- The string is neither null nor empty.

An error occurs if:
- The SYSSTRINGS table is used but contains no information about the pair of CCSIDs and DB2 cannot do the conversion through z/OS support for Unicode.
- A character of the string cannot be converted and the operation is assignment to a column or to a host variable that has no indicator variable. For example, a DBCS character cannot be converted to a variable with an SBCS CCSID.

A warning occurs if:
- A character of the string is converted to a substitution character. A *substitution character* is the character that is used when a character of the source character set is not part of the target character set. For example, assuming an EBCDIC target character set, if the source character set includes Katakana characters and the target character set does not, a Katakana character is converted to the EBCDIC SUB X'3F'.
- A character of the string cannot be converted and the operation is assignment to a variable that has an indicator variable. For example, a DBCS character cannot be converted if the variable has an SBCS CCSID. In this case, the string is not assigned to the variable and the indicator variable is set to -2.

Datetime assignments

A value that is assigned to a date, time, or timestamp column, variable, or parameter must be a valid string representation of a date, a time, or a timestamp.

A value that is assigned to a DATE column, a DATE variable, or a DATE must be a date or a valid string representation of a date. A date can be assigned only to the following items:
- a DATE column
- a character-string column
- a character-string variable
- a date global variable

A value that is assigned to a TIME column, a TIME variable, or a TIME parameter must be a time or a valid string representation of a time. A time can be assigned only to the following items:
- a TIME column
- a character-string column
- a character-string variable
- a time global variable

A value that is assigned to a timestamp column, a timestamp variable, or a timestamp parameter must be a timestamp, a TIMESTAMP constant, or a valid string representation of a timestamp. A timestamp can be assigned only to the following items:

- a TIMESTAMP column
- a character-string or graphic-string column
- a timestamp variable
- a character-string or graphic-string variable
- a timestamp global variable

A valid string representation of a datetime value must not be a BLOB, CLOB, or DBCLOB. A datetime value cannot be assigned to a column that has a field procedure. If the timestamp precision of the target is less than the timestamp precision of the assigned value, the excessive fractional seconds are truncated.

When a datetime value is assigned to a character-string variable or column, it is converted to its string representation. Leading zeros are not omitted from any part of the date, time, or timestamp. The required length of the target varies depending on the format of the string representation. If the length of the fixed length character-string target is greater than required, it is padded on the right with blanks. If the length of the target is less than required, the result depends on the type of datetime value involved, and the type of the target.

When a datetime value is assigned to a timestamp variable or column, it is converted to the target timestamp data type. If the source data type is not the same as the target data type, the source value is implicitly cast to the target data type. DB2 might implicitly cast data types during assignments that involve a distinct type.

- If the target is not a variable and has a character-string or graphic-string data type (except for BLOB, CLOB, or DBCLOB), truncation is not allowed. The length of the column must be at least the following values:
  - 10 for a **DATE**
  - 8 for a **TIME**
  - 19 for a **TIMESTAMP WITHOUT TIME ZONE** with a precision of `0, 20+`*p* with precision of *p*
  - Sufficient to include the time zone (truncation is not allowed), for a **TIMESTAMP WITH TIME ZONE**
- When the target is a variable, the following rules apply:
  - **For a DATE:** The length of the variable must not be less than 10.
  - **For a TIME:** If the USA format is used, the length of the variable must not be less than 8. This format does not include seconds.

    If the ISO, EUR, or JIS format is used, the length of the variable must not be less than 5. If the length is 5, 6, or 7, the seconds part of the time is omitted from the result and SQLWARN1 is set to 'W'. In this case, the seconds part of the time is assigned to the indicator variable if one is provided, and, if the length is 6 or 7, the value is padded with blanks so that it is a valid string representation of a time.
  - **For a timestamp:** The length of the variable must not be less than 19. If the source is TIMESTAMP WITH TIME ZONE, the length of the variable must be sufficient to include the time zone, truncation is not allowed.

- If the length is between 19 and 31, the timestamp is truncated like a string, which causes the omission of one or more digits of the fractional seconds part of a timestamp.
- If the length is 20, the trailing decimal point is excluded so that the value is a valid string representation of a timestamp with precision 0.

Row ID assignments

A row ID value can be assigned only to a column, parameter, or host variable with a row ID data type.

For the value of the ROWID column, the column must be defined as GENERATED BY DEFAULT and the column must have a unique, single-column index. The value that is specified for the column must be a valid row ID value that was previously generated by DB2.

XML assignments

XML data can be assigned to a column, but when the target is not a column, the XML data type can only be assigned to another XML data type.

When the target is a column (for example, data change statements), the source can be the XML data type, or CHAR, VARCHAR, CLOB, GRAPHIC, VARGRAPHIC, DBCLOB, BINARY, VARBINARY, or BLOB data types. When the source is not XML data, the source is implicitly parsed as if the XMLPARSE function is invoked with the STRIP WHITESPACE option. If the source data is graphic data, the encoding scheme must be Unicode.

All other data types cannot be assigned to a target of the XML data type.

User-defined type assignments

User-defined type assignments include distinct type assignments and array assignments.

Distinct type assignments

The rules that apply to the assignments of distinct types to host variables are different than the rules for all other assignments that involve distinct types.

Assignments to host variables: The assignment of distinct type to a host variable is based on the source data type of the distinct type. Therefore, the value of a distinct type is assignable to a host variable only if the source data type of the distinct type is assignable to the host variable.

Example: Assume that distinct type AGE was created with the following SQL statement:

```
CREATE TYPE AGE AS SMALLINT;
```

When the statement was executed, DB2 also generated these cast functions:

```
AGE (SMALLINT) RETURNS AGE
AGE (INTEGER) RETURNS AGE
SMALLINT (AGE) RETURNS SMALLINT
```

Next, assume that column STU_AGE was defined in table STUDENTS with distinct type AGE. Now, consider this valid assignment of a student's age to host variable *HV_AGE*, which has an INTEGER data type:

```
SELECT STU_AGE INTO :HV_AGE FROM STUDENTS WHERE STU_NUMBER = 200;
```

The distinct type value is assignable to host variable *HV_AGE* because the source data type of the distinct type (SMALLINT) is assignable to the host variable (INTEGER). If distinct type AGE had been based on a character data type such as CHAR(5), the above assignment would be invalid because a character type cannot be assigned to an integer type.

Assignments other than to host variables: A distinct type can be the source or target of an assignment. Assignment is based on whether the data type of the value to be assigned is castable to the data type of the target. (Table 15 on page 112 shows which casts are supported when a distinct type is involved). Therefore, a distinct type value can be assigned to any target other than a host variable when:

- The target of the assignment has the same distinct type, or
- The distinct type is castable to the data type of the target

Any value can be assigned to a distinct type when:

- The value to be assigned has the same distinct type as the target, or
- The data type of the assigned value is castable to the target distinct type

Example: Assume that the source data type for distinct type AGE is SMALLINT:

```
CREATE TYPE AGE AS SMALLINT;
```

Next, assume that two tables TABLE1 and TABLE2 were created with four identical column descriptions:

```
AGECOL   AGE
SMINTCOL SMALLINT
INTCOL   INTEGER
DECCOL   DEC(6,2)
```

Using the following SQL statement and substituting various values for X and Y to insert values into various columns of TABLE1 from TABLE2, the following table shows whether the assignments are valid. DB2 uses assignment rules in this INSERT statement to determine if X can be assigned to Y.

```
INSERT INTO TABLE1 (Y)
     SELECT X FROM TABLE2;
```

Table 24. Assessment of various assignments for example INSERT statement

X (column in TABLE2)	Y (column in TABLE1)	Valid	Reason
AGECOL	AGECOL	Yes	Source and target are same distinct type
SMINTCOL	AGECOL	Yes	SMALLINT can be cast to AGE
INTCOL	AGECOL	Yes	INTEGER can be cast to AGE (because AGE's source type is SMALLINT)
DECCOL	AGECOL	No	DECIMAL cannot be cast to AGE
AGECOL	SMINTCOL	Yes	AGE can be cast to its source type of SMALLINT
AGECOL	INTCOL	No	AGE cannot be cast to INTEGER
AGECOL	DECCOL	No	AGE cannot be cast to DECIMAL

Array type assignments

An array value can only be assigned to a variable or parameter with a compatible user-defined array type.

The following values can be assigned to an array variable:

- The null value.
- The value of an array with a user-defined array type, where the source and target arrays have the same user-defined array type. The source value can be an array variable, an invocation of the TRIM_ARRAY function, an invocation of the ARRAY_DELETE function, or a CAST specification. The value of an ordinary array variable or parameter can only be assigned to an ordinary array target. The value of an associative array variable or parameter can only be assigned to an associative array target.
- The value of an array without a user-defined array type. The result of an invocation of aggregate built-in function ARRAY_AGG or of an array constructor is an array without an associated user-defined data type.

  For an assignment with a FETCH statement, the elements in the source array value must have the same data type as the elements of the user-defined array type of the target array. The index values of the source array value must have the same data type as the index of the user-defined array type of the target array.

  For an assignment that is the result of a statement other than FETCH, the source array value is implicitly cast to the target array type.

Assignment of a value to an array element might affect the cardinality of the array, and might result in initializing other new array elements with the null value. Suppose that A is the target array variable, *c* is the cardinality of array A, *idx* is an expression that is used as the array index, and SV is the source value. DB2 assigns the values to the elements of the array as follows:

- If array A is the null value, A is set to an empty array.
- If A is an ordinary array:
  - If *idx* is less than or equal to *c*, the value in the element of A with array index *idx* is replaced by the value of SV.
  - If *idx* greater than *c*:
    - Each element of A with array index *i*, for every *i* that is greater than *c* and less than *idx*, is set to the null value.
    - The element of A with array index *idx* is set to the value of SV.
    - The cardinality of A is set to *idx*.
- If A is an associative array:
  - If *idx* matches an existing value of the array index for A, the value of the element with array index *idx* is replaced by the value of SV.
  - If *idx* does not match an existing value of the array index for A:
    - The element of A with array index *idx* is set to the value of SV.
    - The cardinality of A is incremented by 1.

The following values can be assigned to an element of an array variable:

- The null value
- The value of an expression, where the data type of the expression is assignable to the data type of the elements in the target array

Examples

Example: Assigning an array to another array

Suppose that arrays PHONELIST and HOMEPHONELIST are defined with the same user-defined array type named PLIST. PLIST is defined with

VARCHAR(12) elements. The following statement assigns the values of the HOMEPHONELIST array to the PHONELIST array:

```
SET PHONELIST = HOMEPHONELIST;
```

Example: Assigning elements of an array to another array

Suppose that array V is defined with user-defined type MYARRAY. The following statement assigns the values 1, 2, and 3 to array V using an array constructor.

```
SET V = ARRAY[1,2,3];
```

This statement is equivalent to the following statement:

```
SET V = CAST(ARRAY[1,2,3] AS MYARRAY);
```

Example: Assigning values from a column to an array

Suppose that array V is defined with user-defined type MYARRAY. The following statement assigns the values from the column C1 in table T to array V using the ARRAY_AGG function.

```
SELECT ARRAY_AGG(C1) INTO V FROM T;
```

This statement is equivalent to the following statement:

```
SELECT CAST(ARRAY_AGG(C1) AS MYARRAY) INTO V FROM T;
```

Assignments to LOB locators

When a LOB locator is used, it can refer only to LOB data. If a LOB locator is used for the first fetch of a cursor, LOB locators must be used for all subsequent fetches.

Numeric comparisons

Numbers are compared algebraically, that is, with regard to sign. For example, -2 is less than +1. When numbers of different data types are compared, certain rules are in effect as to how the comparison is performed.

If one number is an integer and the other is decimal, the comparison is made with a temporary copy of the integer, which has been converted to decimal.

When decimal numbers with different scales are compared, the comparison is made with a temporary copy of one of the numbers that has been extended with trailing zeros so that its fractional part has the same number of digits as the other number.

If one number is double precision floating-point and the other is integer, decimal, or single precision floating-point, the comparison is made with a temporary copy of the other number which has been converted to double precision floating-point. However, if a single precision floating-point number is compared with a floating-point constant, the comparison is made with a single precision form of the constant.

Two floating-point numbers are equal only if the bit configurations of their normalized forms are identical.

If one number is DECFLOAT and the other number is integer, decimal, single precision floating-point, or double precision floating-point, the comparison is made with a temporary copy of the other number which has been converted to DECFLOAT.

If one number if DECFLOAT(16) and the other number is DECFLOAT(34), the DECFLOAT(16) value is converted to DECFLOAT(34) before the comparison.

Additionally, the DECFLOAT data type supports both positive and negative zero. Positive and negative zero have different binary representations, but the equal (=) predicate will return true for comparisons of positive and negative zero.

The functions, COMPARE_DECFLOAT and TOTALORDER can be used to perform comparisons at a binary level. For example, for a comparison of 2.0<>2.00.

The DECFLOAT data type also supports the specification of negative and positive NaN (quiet and signaling), and negative and positive infinity. From an SQL perspective, infinity = infinity, NaN = NaN, and sNaN = sNaN.

The following rules are the comparison rules for these special values:

- Infinity compares equal only to infinity of the same sign (positive or negative)
- NaN compares equal only to NaN of the same sign (positive or negative)
- sNaN compares equal only to sNaN of the same sign (positive or negative)

The ordering among the different special values is as follows: `-NAN < -SNAN < -INFINITY < 0 < INFINITY < SNAN <NAN`

String comparisons

String comparisons can occur with binary string, character strings, and graphic strings.

Binary string comparisons

Binary string comparisons are always performed according to the binary values.

Two binary strings are equal only if the lengths of the two strings are identical. If the strings are equal up to the length of the shorter string length, the shorter string is considered less than the longer string even when the remaining bytes in the longer string are hexadecimal zeros. This is illustrated in the following table:

Table 25. Binary string comparison where one operand is longer because of hexadecimal zeros

Hexadecimal value of the first operand	relationship	Hexadecimal value of the second operand
X'4100'	<	X'410000'
X'4100'	<	X'42'
X'4100'	=	X'4100'
X'4100'	>	X'41'
X'4100'	>	X'400000'

Binary strings cannot be compared to character strings (even FOR BIT DATA) unless the character string is cast to a binary string.

Character and graphic string comparisons

Two strings are compared by comparing the corresponding bytes of each string. If the strings do not have the same length, the comparison is made with a temporary copy of the shorter string that has been padded on the right with blanks so that it has the same length as the other string.

Two strings are equal if they are both empty or if all corresponding bytes are equal. An empty string is equal to a blank string. If two strings are not equal, their relationship (that is, which has the greater value) is determined by the comparison of the first pair of unequal bytes from the left end of the strings. This comparison is made according to the collating sequence associated with the encoding scheme of the data. For ASCII data, characters A through Z (both upper and lowercase) have a greater value than characters 0 through 9. For EBCDIC data, characters A through Z (both upper and lowercase) have a lesser value than characters 0 through 9.

Varying-length strings with different lengths are equal if they differ only in the number of trailing blanks. In operations that select one value from a collection of such values, the value selected is arbitrary. The operations that can involve such an arbitrary selection are DISTINCT, MAX, MIN, and references to a grouping column. See the description of GROUP BY for further information about the arbitrary selection involved in references to a grouping column.

Related concepts:

Objects with different CCSIDs in the same SQL statement (DB2 Internationalization Guide)

String comparisons with field procedures:

The rules for string comparisons with field procedures depend on the values being compared.

If a column with a field procedure is compared with the value of a variable or a constant, the variable or constant is encoded by the field procedure before the comparison is made. If the comparison operator is LIKE, the variable or constant is not encoded and the column value is decoded.

If a column with a field procedure is compared with another column, that column must have the same field procedure and both columns must have the same CCSID set. The comparison is performed on the encoded form of the values in the columns. If the encoded values are numeric, their data types must be identical; if they are strings, their data types must be compatible.

If two encoded strings of different lengths are compared, the shorter is temporarily padded with encoded blanks so that it has the same length as the other string.

In a CASE expression, if a column with a field procedure is used as the *result-expression* in a THEN or ELSE clause, all other columns that are used as *result-expressions* must have the same field procedure. Otherwise, no column used in a *result-expression* can name a field procedure.

Datetime comparisons

A date, time, or timestamp value can be compared with another value of the same data type, a datetime constant of the same data type, or with a string representation of a value of that data type. Additionally, a TIMESTAMP WITHOUT TIME ZONE value can be compared with a TIMESTAMP WITH TIME ZONE value.

All comparisons are chronological, which means the further a point in time is from January 1, 0001, the greater the value of that point in time. The time 24:00:00 compares greater than the time 00:00:00.

Comparisons that involve TIME values and string representations of time values always include seconds. If the string representation omits seconds, zero seconds are implied.

Comparisons that involve timestamp values are evaluated according to the following rules:

- When comparing timestamp values with different precisions, the higher precision is used for the comparison and any missing digits for fractional seconds are assumed to be zero.
- When comparing a TIMESTAMP WITH TIME ZONE value to a TIMESTAMP WITHOUT TIME ZONE value, the TIMESTAMP WITHOUT TIME ZONE value is cast to TIMESTAMP WITH TIME ZONE before the comparison is made.
- When comparing two TIMESTAMP WITH TIME ZONE values, the comparison is made using the UTC representations of the values. Two TIMESTAMP WITH TIME ZONE values are considered equal if they represent the same instance in UTC, regardless of the time zone offsets that are stored in the values. For example, '1999-04-15-08.00.00-08:00' (8:00 a.m. Pacific Standard Time) is the same as '1999-04-15-11.00.00-05:00' (11:00 a.m. Eastern Standard Time).
- When comparing a timestamp value with a string representation of a timestamp, the string representation is first converted to a the data type of the timestamp operand. With the except that the converted value has a precision of 12. If the timestamp operand is TIMESTAMP WITHOUT TIME ZONE, the string must not contain a specification of time zone.
- Timestamp comparisons are chronological without regard to representations that might be considered equivalent. For example, the following predicate is true:

  ```
  TIMESTAMP('1990-02-23-00.00.00') > '1990-02-22-24.00.00'
  ```

Example 1: Table TABLE1 has 2 columns: C1, which is defined as TIMESTAMP WITH TIME ZONE; and C2, which is defined as TIMESTAMP WITHOUT TIME ZONE:

```
CREATE TABLE TABLE1 (C1 TIMESTAMP WITH TIME ZONE, C2 TIMESTAMP);
```

A row is inserted into the table with the following INSERT statement. The input values are provided by character-string representations of a timestamp with a time zone.

```
INSERT INTO TABLE1 VALUES ( '2007-11-05-08.00.00-08:00', '2007-11-05-08.00.00');
```

Assuming that the implicit time zone is `-5:00`, the following SELECT statement will not return any rows. The string representation of the TIMESTAMP WITHOUT TIME ZONE value is cast to a TIMESTAMP WITH TIME ZONE value, which results in a timestamp with time zone value of `'2007-11-05-08.00.00-05:00'` for column C2. The comparison predicate is false because the two values are not equal.

```
SELECT 1 FROM TABLE1 WHERE C1 = C2;
```

Example 2: When a TIMESTAMP WITHOUT TIME ZONE value is compared with a string representation of a TIMESTAMP WITHOUT TIME ZONE or a TIMESTAMP WITH TIME ZONE value, the string representation is cast to TIMESTAMP WITHOUT TIME ZONE (regardless of whether the string contains a time zone). The comparison is performed using the two TIMESTAMP WITHOUT TIME ZONE values. Assume that *string_hv* contains a timestamp with time zone value of `'2007-11-05-08.00.00-08:00'`. The string value is cast to a TIMESTAMP WITHOUT TIME ZONE value of `'2007-11-05-08.00.00'`, which is compared with

the value that is stored in column C2. The following SELECT statement returns a single row because a row exists in the table with a timestamp without time zone value of '2007-11-05-08.00.00'.

```
SELECT 1 FROM TABLE1 WHERE C2 = :string_hv;
```

Row ID comparisons

A value with a row ID type can only be compared to another row ID value.

The comparison of the row ID values is based on their internal representations. The maximum number of bytes that are compared is 17 bytes, which is the number of bytes in the internal representation. Therefore, row ID values that differ in bytes beyond the 17th byte are considered to be equal.

XML comparisons

The XML data type cannot be directly compared to any data type, including the XML data type. The method for doing comparison is through the use of the XMLEXISTS predicate.

Conversion rules for comparisons

When two strings are compared, one of the strings is first converted, if necessary, to the coded character set of the other string. Conversion is necessary only if certain rules apply.

Conversion is necessary only if all of the following are true:

- The CCSIDs of the two strings are different.
- Neither CCSID is X'FFFF' (neither string is defined as a binary string).
- The string selected for conversion is neither null nor empty.
- The following conversion tables (Table 27 on page 140 or Table 28 on page 140) indicate when conversion is necessary.

The string selected for conversion depends on the type of the operands. For the purpose of CCSID determination, string expressions in a statement are divided into 6 types, as described in the following table.

Table 26. Operand types

Type of operand	CCSID of the operand type
Columns	CCSID from the containing table
String constants	CCSID associated with the application encoding scheme. For dynamic statements, this is the CURRENT APPLICATION ENCODING SCHEME special register. For static statements, this is the ENCODING bind option or the APPLICATION ENCODING SCHEME option of the CREATE PROCEDURE or ALTER PROCEDURE statement for native SQL procedures..
Special registers	CCSID associated with the application encoding scheme. For dynamic statements, this is the CURRENT APPLICATION ENCODING SCHEME special register. For static statements, this is the ENCODING bind option or the APPLICATION ENCODING SCHEME option of the CREATE PROCEDURE or ALTER PROCEDURE statement for native SQL procedures.
Host variables	CCSID specified in the DECLARE VARIABLE statement, associated with the application encoding scheme, or specified in SQLDAID or SQLDA
Global variables	CCSID of UNICODE

Table 26. Operand types (continued)

Type of operand	CCSID of the operand type
Derived value based on a column	CCSID derived from the source of the derived value. A derived value based on a column is an expression whose source is directly or indirectly based on columns. The CCSID of such an expression is the CCSID derived from its source. For example: • The CCSID of `SUBSTR(column_1, 5, length(column_2))` is the CCSID of column_1. Note that the CCSID of column_2 has no influence on the CCSID of SUBSTR. • The CCSID of `column_1 || 'ABC'` is the CCSID of column_1, derived from the rules described in Table 27 on page 140. • The CCSID of `column_1 || GX'42C1'` is the DBCS CCSID from the CCSID set of column_1, derived from the rules described in Table 27 on page 140 and Table 28 on page 140. • The CCSID of `COALESCE(EBCDIC_column_1, ASCII_column_1)` is the UNICODE CCSID, derived from the rules described in Table 27 on page 140. • The CCSID of `CAST(string_column_1 AS GRAPHIC(10))` is the DBCS CCSID from the CCSID set of string_column_1. • The CCSID of `CAST(EBCDIC_string_column_1 AS VARCHAR(10) CCSID UNICODE)` is the UNICODE CCSID derived from the rules described in Table 27 on page 140. • The CCSID of `CASE WHEN(1=1) THEN '1' ELSE ASCII_column_1 END` is the CCSID of ASCII_column_1, derived from the rules described in Table 27 on page 140. • The CCSID of `CASE WHEN(1=1) THEN EBCDIC_column_1 ELSE ASCII_column_1 END` is the UNICODE CCSID derived from the rules described in Table 27 on page 140. • The CCSID of a scalar fullselect (`SELECT column_1 FROM table_1`) is the CCSID of column_1.
Derived value not based on a column	CCSID derived from the source of the derived value. A derived value not based on a column is an expression whose source is not directly or indirectly based on any column. The CCSID of such an expression is the CCSID derived from its source. • For example, the CCSID of `SUBSTR('ABDC', 1, length('AB"))` is the CCSID of the string constant 'ABCD'. Note that the CCSID of column_1 has no influence on the CCSID of SUBSTR. • the CCSID of `user_defined_function1(column1)` is the output CCSID defined by user_defined_function1. • the CCSID of the cast function of distinct type, shape, is the CCSID of distinct type, shape. • the CCSID of `CURRENT SQLID || UX'0041'` is the UNICODE DBCS CCSID, derived from the rules described in Table 27 on page 140 and Table 28 on page 140. • the CCSID of `CAST('abc' as CHAR(10) CCSID UNICODE)` is the UNICODE CCSID.

The following table shows which operand supplies the target CCSID set when the comparison is part of an SQL statement involving multiple tables with different CCSID sets.

Table 27. Operand that supplies the CCSID for character conversion

	Second operand					
First operand	**Column value**	**String constant**	**Special register**	**Host variable**	**Derived value based on a column**	**Derived value not based on a column**
Column value	1	first operand	first operand	first operand, 2	1	first operand
String constant	second operand	1	1	1	second operand	1
Special register	second operand	1	1	1	second operand	1
Host variable	second operand	1	1	1	second operand, 2	1
global variable	second operand					
Derived value based on a column	1	first operand	first operand	first operand, 2	1	first operand
Derived value not based on a column	second operand	1	1	1	second operand	1

Note:

1. If the CCSID sets are different, both operands are converted, if necessary, to Unicode. SBCS and Mixed are converted to UTF-8. DBCS is converted to UTF-16. See the next table to determine which operand supplies the CCSID for character conversion.
2. If the host variable is Unicode graphic, and the value of the field MIXED DATA on installation panel DSNTIPF is NO, the column or the derived value based on a column supplies the target CCSID set.

The following table shows which operand is selected for conversion when both operands are based on a column or are not based on a column as represented in the previous table.

Table 28. Operand that supplies the CCSID for character conversion when both operands are based or not based on a column

	Second operand		
First operand	**SBCS data**	**Mixed data**	**DBCS data**
SBCS data		second operand[1]	second operand
Mixed data	first operand[1]		second operand
DBCS data	first operand	first operand	

Note:

1. For ASCII and EBCDIC data, the conversion depends on the value of the field MIXED DATA on installation panel DSNTIPF at the DB2 that does the comparison. If MIXED DATA = YES, the SBCS operand is converted to MIXED. If MIXED DATA = NO, the MIXED operand is converted to SBCS

For example, assume a comparison of the form:

```
string-constant-SBCS =derived-value-not-based-on-column-DBCS
```

Assume that the operands have different encoding schemes. First look at Table 27 on page 140. The relevant table entry is in the third row and second column. The value for this entry shows that if the CCSID sets are different, the operands are converted to Unicode. The first operand (string-constant-SBCS) is converted to UTF-8 (Mixed) if it is not already Unicode. In addition, the second operand (derived-value-not-based-on-column-DBCS) is converted to UTF-16 (Unicode DBCS) if necessary. After the operands have been converted to Unicode, Table 28 on page 140 is used to determine which operand supplies the specific CCSID for the conversion. The relevant table entry is in the second row and third column. It indicates that the second operand (derived-value-not-based-on-column-DBCS) determines the CCSID because DBCS data takes precedence over Mixed data.

An error occurs if a character of the string cannot be converted, the SYSSTRINGS table is used but contains no information about the pair of CCSIDs of the operands being compared, or DB2 cannot do the conversion through z/OS support for Unicode. A warning occurs if a character of the string is converted to a substitution character.

A derived value based on a column is an expression that includes columns that affects the result CCSID of the expression. For example, in the expression `COL1||'abc'`, COL1 determines the result CCSID. Therefore, the expression `COL1||'abc'` is considered to be a derived value based on a column that continues to give the column precedence in any further comparisons. The expression `CASE WHEN COL1 > 1 THEN 'abc' ELSE 'def' END` contains a column that does not affect the result CCSID of the expression and is therefore not considered to be a derived value based on a column.

The following table defines which expressions are considered to be a derived value based on a column.

Table 29. Derived values based on a column

Expression	Condition
expression1 || *expression2*	*expression1* or *expression2* is a column or a derived value based on a column
CASE *when-clause* THEN *result-expression* ELSE *result-expression* END	any *result-expression* is a *string-expression* that is a column or derived value based on a column
CAST(*expression* as *data-type*)	*expression* is a *string-expression* that is a column or a derived value based on a column and *data-type* is a string data type
Scalar-fullselect: (SELECT *expression* FROM *table*)	*expression* is a *string-expression* that is a column or a derived value based on a column and *data-type* is a string data type

When a statement contains multiple CCSID sets, if the length of one of the strings changes after CCSID conversion, the string becomes a varying-length string. That is, the data type becomes VARCHAR, CLOB, VARGRAPHIC, or DBCLOB. The following table shows the worse case resulting lengths of CCSID conversion, where X is length in bytes.

Table 30. Worst case result length of CCSID conversion, where X represents LENGTH(string in bytes)

		To CCSID								
		EBCDIC			ASCII			Unicode		
From CCSID		SBCS	Mixed	DBCS	SBCS	Mixed	DBCS	SBCS	UTF-8	UTF-16
EBCDIC	SBCS	X	X	X*2[1]	X	X	X*2[1]	X[1]	X*3	X*2
	Mixed	X	X	X*2[1]	X	X	X*2[1]	X[1]	X*3	X*2
	DBCS	X*0.5[1]	X+2	X	X*0.5[1]	X	X	X*0.5	X*1.5	X
ASCII	SBCS	X	X	X*2[1]	X	X	X*2[1]	X[1]	X*3	X*2
	Mixed	X	X*1.8	X*2[1]	X	X	X*2[1]	X[1]	X*3	X*2
	DBCS	X*0.5[1]	X+2	X	X*0.5[1]	X	X	X*0.5	X*1.5	X
Unicode	SBCS	X	X	X*2	X	X	X*2	X	X	X*2
	UTF-8	X	X*1.25	X	X	X	X	X	X	X*2
	UTF-16	X*0.5	X+2	X	X*0.5	X	X	X*0.5	X*1.5	X

Note:

1. Because of the high probability of data loss, IBM does not provide conversion tables for this combination of two CCSIDs and data subtypes.

User-defined type comparisons

User-defined type comparisons include distinct type comparisons and array comparisons.

Distinct type comparisons

A value with a distinct type can only be compared to another value with exactly the same type because distinct types have strong typing, which means that a distinct type is compatible only with its own type.

To compare a distinct type to a value with a different data type, the distinct type value must be cast to the data type of the comparison value or the comparison value must be cast to the distinct type. For example, because constants are built-in data types, a constant can be compared to a distinct type value only if it is first cast to the distinct type or vice versa.

The following table shows examples of valid and invalid comparisons, assuming the following SQL statements were used to define two distinct types AGE_TYPE and CAMP_DATE and table CAMP_ROSTER table.

```
CREATE TYPE AGE_TYPE AS INTEGER;
CREATE TYPE CAMP_DATE AS DATE;
CREATE TABLE CAMP_ROSTER
   ( NAME               VARCHAR(20),
     ATTENDEE_NUMBER    INTEGER NOT NULL,
     AGE                AGE_TYPE,
     FIRST_CAMP_DATE    CAMP_DATE,
     LAST_CAMP_DATE     CAMP_DATE,
     BIRTHDATE          DATE);
```

Table 31. Examples of valid and invalid comparisons involving distinct types

SQL statement	Valid	Reason
Distinct types with distinct types		
`SELECT * FROM CAMP_ROSTER` `  WHERE FIRST_CAMP_DATE < LAST_CAMP_DATE;`	Yes	Both values are the same distinct type.

Table 31. Examples of valid and invalid comparisons involving distinct types (continued)

SQL statement	Valid	Reason
Distinct types with columns of the same source data type		
`SELECT * FROM CAMP_ROSTER` `  WHERE AGE > ATTENDEE_NUMBER;`	No	A distinct type cannot be compared to integer.

`SELECT * FROM CAMP_ROSTER`
`  WHERE INTEGER(AGE) > ATTENDEE_NUMBER;`
`SELECT * FROM CAMP_ROSTER`
`  WHERE CAST(AGE AS INTEGER) > ATTENDEE_NUMBER;`
	Yes	The distinct type is cast to an integer, making the comparison of two integers.

`SELECT * FROM CAMP_ROSTER`
`  WHERE AGE > AGE_TYPE(ATTENDEE_NUMBER);`
`SELECT * FROM CAMP_ROSTER`
`  WHERE AGE > CAST(ATTENDEE_NUMBER as AGE_TYPE);`
	Yes	Integer ATTENDEE_NUMBER is cast to the distinct type AGE_TYPE, making both values the same distinct type.

Distinct types with constants		
`SELECT * FROM CAMP_ROSTER` `  WHERE AGE IN (15,16,17);`	No	A distinct type cannot be compared to a constant.
`SELECT * FROM CAMP_ROSTER` `  WHERE INTEGER(AGE) IN (15,16,17);`	Yes	The distinct type is cast to the data type of constants, making all the values in the comparison integers.

`SELECT * FROM CAMP_ROSTER`
`  WHERE AGE IN`
`  (AGE_TYPE(15),AGE_TYPE(16),AGE_TYPE(17));`
	Yes	Constants are cast to distinct type AGE_TYPE, making all the values in the comparison the same distinct type.

`SELECT * FROM CAMP_ROSTER` `  WHERE FIRST_CAMP_DATE > '06/12/99';`	No	A distinct type cannot be compared to a constant.

`SELECT * FROM CAMP_ROSTER`
`  WHERE FIRST_CAMP_DATE >`
`  CAST('06/12/99' AS CAMP_DATE);`
	No	The string constant '06/12/99', a VARCHAR data type, cannot be cast directly to distinct type CAMP_DATE, which is based on a DATE data type. As illustrated in the next row, the constant must be cast to a DATE data type and then to the distinct type.

`SELECT * FROM CAMP_ROSTER`
`  WHERE FIRST_CAMP_DATE >`
`  CAST(DATE('06/12/1999') AS CAMP_DATE);`
	Yes	The string constant '06/12/99' is cast to the distinct type CAMP_DATE, making both values the same distinct type. To cast a string constant to a distinct type that is based on a DATE, TIME, or TIMESTAMP data type, the string constant must first be cast to a DATE, TIME, or TIMESTAMP data type.

Distinct types with host variables		
`SELECT * FROM CAMP_ROSTER` `  WHERE AGE BETWEEN :HV_INTEGER AND :HV_INTEGER2;`	No	The host variables have integer data types. A distinct type cannot be compared to an integer.

`SELECT * FROM CAMP_ROSTER`
`  WHERE AGE`
`  BETWEEN CAST(:HV_INTEGER AS AGE_TYPE)`
`  AND AGE_TYPE(:HV_INTEGER2);`
	Yes	The host variables are cast to distinct type AGE_TYPE, making all the values the same distinct type.

`SELECT * FROM CAMP_ROSTER` `  WHERE FIRST_CAMP_DATE > :HV_VARCHAR;`	No	The host variable has a VARCHAR data type. A distinct type cannot be compared to a VARCHAR.

Table 31. Examples of valid and invalid comparisons involving distinct types (continued)

SQL statement	Valid	Reason

`SELECT * FROM CAMP_ROSTER`
`WHERE FIRST_CAMP_DATE >`
`CAST(DATE(:HV_VARCHAR) AS CAMP_DATE);`
	Yes	The host variable is cast to the distinct type CAMP_DATE, making both values the same distinct type. To cast a VARCHAR host variable to a distinct type that is based on a DATE, TIME, or TIMESTAMP data type, the host variable must first be cast to a DATE, TIME, or TIMESTAMP data type.

Array type comparisons

Comparisons of array values are not supported.

Elements of arrays can be compared based on the comparison rules for the data types of the elements of the arrays.

Rules for result data types

Rules that are applied to the operands of an operation determine the data type of the result. Certain rules apply in certain situations and apply depending on the data type of operands.

The rules apply to:

- Corresponding columns in set operations (UNION, INTERSECT, or EXCEPT)
- Result expressions of a CASE expression
- Arguments of the scalar functions COALESCE, IFNULL, MAX, and MIN
- Expression values of the IN list of an IN predicate
- Expression values for the elements in an array constructor
- Expression values for the arguments for a collection-derived table (UNNEST specification)
- Arguments of a BETWEEN predicate, except if the data types of all operands are numeric
- Arguments for the aggregation group ranges in OLAP specifications

For the result data type of expressions that involve the operators '/', '*', '+' and '-', see “Expressions with arithmetic operators” on page 243.

For the result data type of expressions that involve the CONCAT operator, see “Expressions with the concatenation operator” on page 250.

Evaluation of the operands of an operation determines the data type of the result. If an operation has more than one pair of operands, DB2 determines the result type of the first pair, uses this result type with the next operand to determine the next result type, and so on. The last intermediate result type and the last operand determine the result type of the operation.

With the exception of the COALESCE function, the result of an operation can be null unless the operands do not allow nulls.

If the data type and attributes of any operand column are not the same as those of the result, the operand column values are converted to conform to the data type and attributes of the result. The conversion operation is exactly the same as if the values were assigned to the result. For example:

- If one operand column is CHAR(10), and the other operand column is CHAR(5), the result is CHAR(10), and the values derived from the CHAR(5) column are padded on the right with five blanks.
- If the whole part of a number cannot be preserved then an error is returned.

Related concepts:

"Conversion rules for comparisons" on page 138

Numeric operands

Numeric types are compatible only with other numeric types.

Table 32. Result data types with numeric operands

One operand	Other operand	Data type of the result
SMALLINT	SMALLINT	SMALLINT
INTEGER	INTEGER	INTEGER
INTEGER	SMALLINT	INTEGER
BIGINT	SMALLINT	BIGINT
BIGINT	INTEGER	BIGINT
BIGINT	BIGINT	BIGINT
DECIMAL(*w*,*x*)	SMALLINT	DECIMAL(*p*,*x*) where *p* = *x*+max(*w*-*x*,5)[1]
DECIMAL(*w*,*x*)	INTEGER	DECIMAL(*p*,*x*) where *p* = *x*+max(*w*-*x*,11)[1]
DECIMAL(*w*,*x*)	BIGINT	DECIMAL(*p*,*x*) where *p* = *x*+max(*w*-*x*,19)[1]
DECIMAL(*w*,*x*)	DECIMAL(y,z)	DECIMAL(*p*,*s*) where *p* = max(*x*,*z*)+max(*w*-*x*,*y*-*z*)[1]*s* = max(*x*,*z*)
REAL	REAL	REAL
REAL	DECIMAL, BIGINT, INTEGER, or SMALLINT	DOUBLE
REAL	BIGINT	DOUBLE
DOUBLE	DOUBLE, REAL, DECIMAL, BIGINT, INTEGER, or SMALLINT	DOUBLE
DECFLOAT(*n*)	SMALLINT	DECFLOAT(*n*)
DECFLOAT(*n*)	INTEGER	DECFLOAT(*n*)
DECFLOAT(*n*)	BIGINT	DECFLOAT(34)
DECFLOAT(*n*)	DECIMAL(<=16,*s*)	DECFLOAT(*n*)
DECFLOAT(*n*)	DECIMAL(>16,*s*)	DECFLOAT (34)
DECFLOAT(*n*)	REAL	DECFLOAT(*n*)
DECFLOAT(*n*)	DOUBLE	DECFLOAT(*n*)
DECFLOAT(*n*)	DECFLOAT(*m*)	DECFLOAT(max(*n*,*m*))

Table 32. Result data types with numeric operands (continued)

One operand	Other operand	Data type of the result

Notes:

1. Precision cannot exceed 31.

Character and graphic string operands

Character and graphic strings are compatible with other character and graphic strings as long as there is a conversion between their corresponding CCSIDs.

Table 33. Result data types with string operands

One operand	Other operand	Data type of the result
CHAR(*x*)	CHAR(*y*)	CHAR(*z*) where *z* = max(*x*,*y*)
GRAPHIC(*x*)	CHAR(*y*)	VARGRAPHIC(*y*) where *y* > maximum length of a graphic
GRAPHIC(*x*)	CHAR(*y*)	GRAPHIC(*z*) where *z* = max(*x*,*y*)
VARCHAR(*x*)	VARCHAR(*y*) or CHAR(*y*)	VARCHAR(*z*) where *z* = max(*x*,*y*)
VARCHAR(*x*)	GRAPHIC(*y*)	VARGRAPHIC(*z*) where *z* = max(*x*,*y*)
VARGRAPHIC(*x*)	VARGRAPHIC(*y*), GRAPHIC(*y*), VARCHAR(*y*), or CHAR(*y*)	VARGRAPHIC(*z*) where *z* = max(*x*,*y*)
CLOB(*x*)	CLOB(*y*), VARCHAR(*y*), or CHAR(*y*)	CLOB(*z*) where *z* = max(*x*,*y*)
CLOB(*x*)	GRAPHIC(*y*) or VARGRAPHIC(*y*)	DBCLOB(*z*) where *z* = max(*x*,*y*)
DBCLOB(*x*)	CHAR(*y*), VARCHAR(*y*), CLOB(y), GRAPHIC(*y*), VARGRAPHIC(*y*), or DBCLOB(*y*)	DBCLOB(*z*) where *z* = max(*x*,*y*)

Character string subtypes are determined as indicated in the following table:

Table 34. Result data types with character string operands

One operand	Other operand	Data type of the result
Bit data	Mixed, SBCS, or bit data	Bit data
Mixed data	Mixed or SBCS data	Mixed data
SBCS data	SBCS data	SBCS data

Binary string operands

Binary strings are compatible with other binary strings. Binary strings include BINARY, VARBINARY, and BLOB.

Table 35. Result data types with binary string operands

One operand	Other operand	Data type of the result
BINARY(*x*)	BINARY(*y*)	BINARY(*z*) where *z* = max(*x*,*y*)

Table 35. Result data types with binary string operands (continued)

One operand	Other operand	Data type of the result
VARBINARY(*x*)	BINARY(*y*) or VARBINARY(*y*)	VARBINARY(*z*) where *z* = max(*x*,*y*)
BLOB(*x*)	BINARY(*y*), VARBINARY(*y*), or BLOB(*y*)	BLOB(*z*) where *z* = max(*x*,*y*)

Datetime operands

A date, time, or timestamp value is compatible with another value of the same type or any string expression that contains a valid string representation of the same type.

A DATE type is compatible with another DATE type or any string expression that contains a valid string representation of a date. A string representation is a value that is a built-in character string data type or graphic string data type. A string representation must not be a CLOB or DBCLOB and must have an actual length that is not greater than 255 bytes. The data type of the result is DATE.

A TIME type is compatible with another TIME type or any string expression that contains a valid string representation of a time. A string representation is a value that is a built-in character string data type or graphic string data type. A string representation must not be a CLOB or DBCLOB and must have an actual length that is not greater than 255 bytes. The data type of the result is TIME.

A timestamp type is compatible with another timestamp type, a timestamp constant, or any string expression that contains a valid string representation of a timestamp. A string representation is a value that is a built-in character string data type or graphic string data type. A string representation must not be a CLOB or DBCLOB and must have an actual length that is not greater than 255 bytes. The data type of the result is a timestamp as determined in the following table.

Table 36. Result data types with datetime operands

One operand	Other operand	Data type of the result
TIMESTAMP(*x*) WITHOUT TIME ZONE	TIMESTAMP(*y*) WITHOUT TIME ZONE	TIMESTAMP(max(*x*,*y*)) WITHOUT TIME ZONE
TIMESTAMP(*x*) WITHOUT TIME ZONE	CHAR(*y*) or VARCHAR(*y*)	TIMESTAMP(*x*) WITHOUT TIME ZONE [1]
TIMESTAMP(*x*) WITH TIME ZONE	TIMESTAMP(*y*) WITH TIME ZONE	TIMESTAMP(max(*x*,y)) WITH TIME ZONE
TIMESTAMP(*x*) WITH TIME ZONE	CHAR(*y*) or VARCHAR(*y*)	TIMESTAMP(*x*) WITH TIME ZONE
TIMESTAMP(*x*) WITH TIME ZONE	TIMESTAMP(*y*) WITHOUT TIME ZONE	TIMESTAMP(max(*x*,*y*)) WITH TIME ZONE

Note: If one operand is TIMESTAMP(*x*) WITHOUT TIME ZONE and the other operand is CHAR(*y*) or VARCHAR(*y*), the result data type is TIMESTAMP(*x*) WITHOUT TIME ZONE even if the string representation contains a time zone.

If both operands are in the same encoding scheme, the result is in that encoding scheme. Otherwise the result is in the application encoding scheme.

Row ID operands

A row ID data type is compatible only with itself. The result has a row ID data type.

XML operands

XML data is compatible only with other XML data. The data type of the result is XML.

Other data types can be treated as an XML data type by using the CAST specification or XMLPARSE functions to cast character, graphic, or binary data to XML data.

Distinct type operands

A distinct type is compatible only with itself. The data type of the result is the distinct type.

Constants

A *constant* (also called a *literal*) specifies a value. Constants are classified as string constants or numeric constants. Numeric constants are further classified as integer, floating-point, decimal, or decimal floating-point. String constants are classified as character, graphic, or binary.

All constants have the attribute NOT NULL. A negative sign in a numeric constant with a value of zero is ignored, except for a decimal floating-point constant.

Constants have a built-in data type. Therefore, an operation that involves a constant and a distinct type requires that the distinct type be cast to the built-in data type of the constant or the constant be cast to the distinct type. For example, see Table 31 on page 142, which contains an example of casting data types to compare a constant to a distinct type.

Integer constants

An *integer constant* specifies an integer as a signed or unsigned number with a maximum of 19 digits that does not include a decimal point.

The data type of an integer constant is large integer if its value is within the range of a large integer. The data type of an integer constant is big integer if its value is outside the range of a large integer, but within the range of a big integer. A constant that is defined outside the range of big integer values is considered a decimal constant.

Examples:

```
64     -15     +100     32767     720176
```

In syntax diagrams, the term *integer* is used for a large integer constant that must not include a sign.

Floating-point constants

A *floating-point constant* specifies a double-precision floating-point number as two numbers separated by an E.

The first number can include a sign and a decimal point. The second number can include a sign but not a decimal point. The value of the constant is the product of the first number and the power of 10 specified by the second number. It must be within the range of floating-point numbers. The number of characters in the constant must not exceed 30. Excluding leading zeros, the number of digits in the first number must not exceed 17 and the number of digits in the second must not exceed 2.

Examples: The following floating-point constants represent the numbers '150', '200000', -0.22, and '500':

```
15E1     2.E5    -2.2E-1     +5.E+2
```

Decimal constants

A *decimal constant* is a signed or unsigned number of no more than 31 digits and either includes a decimal point or is not within the range of binary integers.

The precision is the total number of digits, including those, if any, to the right of the decimal point. The total includes all leading and trailing zeros. The scale is the number of digits to the right of the decimal point, including trailing zeros.

Examples: The following decimal constants have, respectively, precisions and scales of 5 and 2; 4 and 0; 2 and 0; and 23 and 2:

```
025.50   1000.   -15.   +375893333333333333333.33
```

Decimal floating-point constants

A *decimal floating-point constant* specifies a decimal floating-point number as two numbers separated by an E. The first number can include a sign and a decimal point. The second number can include a sign but not a decimal point.

The value of the constant is the product of the first number and the power of 10 specified by the second number. The value must be within the range of DECFLOAT(34). The number of characters in the constant must not exceed 42. Excluding leading zeros, the number of digits in the first number must not exceed 34 and the number of digits in the second number must not exceed 4.

A constant that is specified as two numbers separated by an E is a decimal-floating point constant only is the value is outside the range of a floating-point constant. A constant that is specified as a number that does not contain an E, and has more than 31 digits, is also a decimal-floating point constant.

In addition to numeric constants, the following special values can be used to specify decimal-floating point special values:

- INF or INFINITY - represents infinity
- NAN - represents quiet not-a-number
- SNAN - represents signaling not-a-number

The special values can be any combination of uppercase or lowercase letters and can be preceded by an operational sign (+ or -).

SNAN results in a warning or exception when it is used in a numerical operation; NAN does not. SNAN can be used in non-numerical operations without causing a warning or exception. For example, SNAN can be used in the VALUES list of an insert operation or as a constant used in a comparison in a predicate.

When the special values are used in a predicate, the following order of precedence applies:

```
-NAN < -SNAN < -INFINITY < -0 < 0 < INFINITY < SNAN < NAN
```

Examples: The following decimal floating-point constants represent the numbers 123456789012345678, sNaN, and negative infinity:

```
  12345678901234567E0         SNAN        -INFINITY
```

When one of the special values is used in a context where it could be interpreted as an identifier, such as a column name, cast a string constant that represents the special value to decimal-floating point.

```
CAST ('snan' AS DECFLOAT)
CAST ('INF' AS DECFLOAT)
CAST ('Nan' AS DECFLOAT)
```

Character string constants

A *character string constant* specifies a varying-length character string. There are two forms of character string constant.

- A sequence of characters that starts and ends with a string delimiter, which is either an apostrophe (') or a quotation mark ("). For the factors that determine which is applicable, see "Apostrophes and quotation marks as string delimiters" on page 330. This form of string constant specifies the character string contained between the string delimiters. The number of bytes between the delimiters must not be greater than 32704. The limit of 32704 refers to the length (in bytes) of the UTF-8 representation of the string. If you produced the string in a CCSID other than UTF-8 (for example, an EBCDIC CCSID), the length of the UTF-8 representation might differ from the length of the string's representation in the source CCSID. Two consecutive string delimiters are used to represent one string delimiter within the character string.
- An X followed by a sequence of characters that starts and ends with a string delimiter. This form of a character string constant is also called a *hexadecimal constant*. The characters between the string delimiters must be an even number of hexadecimal digits. The number of hexadecimal digits must not exceed 32704. A hexadecimal digit is a digit or any of the letters A through F. If the MIXED DATA subsystem parameter is set to YES, hexadecimal digits in a hexadecimal constant must be specified in upper case. Otherwise, an error might be returned when SQL statements are processed. Under the conventions of hexadecimal notation, each pair of hexadecimal digits represents a character. A hexadecimal constant allows you to specify characters that do not have a keyboard representation.

Examples:

```
'12/14/1985'    '32'    'DON''T CHANGE'    X'FFFF'    ''
```

The right most string in the example ('') represents an empty character string constant, which is a string of zero length.

A character string constant is classified as mixed data if it includes a DBCS substring. In all other cases, a character string constant is classified as SBCS data. For information about the CCSID that is assigned to the constant, see "Determining the encoding scheme and CCSID of a string" on page 47. A mixed string constant can be continued from one line to the next only if the break occurs between single byte characters. A Unicode string is always considered mixed regardless of the content of the string.

For Unicode, character constants can be assigned to UTF-8 and UTF-16. The form of the constant does not matter. Typically, character string constants are used only with character strings, but they also can be used with graphic UTF-16 data. However, hexadecimal constants are just character data. Thus, hexadecimal constants being used to insert data into UTF-16 data strings should be in UTF-8 format, not UTF-16 format. For example, if you wanted to insert the number 1 into a UTF-16 column, you would use X'31', not X'0031'. Even though X'0031' is a UTF-16 value, DB2 treats it as two separate UTF-8 code points. Thus, X'0031' would become X'00000031'.

Binary string constants

A binary-string constant specifies a varying-length binary string.

A binary-string constant is formed by specifying a BX followed by a sequence of characters that starts and ends with a string delimiter. The characters between the string delimiters must be an even number of hexadecimal digits. The number of hexadecimal digits must not exceed 32704.

A hexadecimal digit is a digit or any of the letters A through F (upper case or lower case). Under the conventions of hexadecimal notation, each pair of hexadecimal digits represents one byte. Note that this representation is similar to the representation of the character-constant that uses the X'' form; however binary-string constant and character-string constant are not compatible and the X'' form can not be used to specify a binary-string constant, just as the BX'' form cannot be used to specify a character-string constant.

Examples of binary-string constants:

```
BX'0000'    BX'C141C242'     BX'FF00FF01FF'
```

Datetime constants

A *datetime constant* is a character string constant of a particular format.

Character-string constants are described under “Character string constants” on page 150.

For information about the valid string formats, see “String representations of datetime values” on page 101.

Typically, character-string constants are used to represent constant datetime values in assignments and comparisons. However, the ANSI/ISO SQL standard form of a datetime constant can be used to specifically denote the constant as a datetime constant instead of a character-string constant. The format for the ANSI/ISO SQL standard datetime constants are as follows:

DATE *string-constant*

string-constant must contain a value that conforms to one of the valid formats for string representations of dates, subject to the following rules:

- leading blanks are not allowed.
- leading zeros can be omitted from the month and day elements of the date. An implicit specification of 0 is assumed for any digit that is omitted.
- leading zeros must be included for the year element of the date.
- trailing blanks can be included.

The data type of the value is DATE.

TIME *string-constant*

string-constant must contain a value that conforms to one of the valid formats for string representations of times, subject to the following rules:

- leading blanks are not allowed.
- leading zeros can be omitted from the hour elements of the time.
- the seconds element of the time can be omitted.
- trailing blanks can be included.
- if the USA format is not used and the minutes and seconds are all zeros, the hour can be 24.
- If the format is USA, the following additional rules apply:
  - the minutes element of the time can be omitted. For example, 1 PM is equivalent to 1:00 PM.
  - the letters A, M, and P can be specified in lowercase.
  - a single blank must precede the AM or PM.
  - the hour must not be greater than 12 and cannot be 0 except when the time is specified as 00:00 AM.

An implicit specification of 0 is assumed for any digit that is omitted.

The correspondence between the USA format and the ISO format (24-hour clock) is as follows:

- 12:01 AM through 12:59 AM correspond to 00.01.00 through 00.59.00
- 01:00 AM through 11:59 AM correspond to 01.00.00 through 11.59.00
- 12:00 PM (noon) through 11:59 PM correspond to 12.00.00 through 23.59.00
- 12:00 AM (midnight) corresponds to 24.00.00
- 00:00 AM (midnight) corresponds to 00.00.00

The data type of the value is TIME.

TIMESTAMP *string-constant*

string-constant must contain a value that conforms to one of the formats listed in the following tables, subject to the following rules:

- leading blanks are not allowed.
- trailing blanks can be included.
- leading zeros can be omitted from the month, day, hour, and time zone hour elements of the timestamp. An implicit specification of 0 is assumed for any digit that is omitted.
- leading zeros must be included for the minute, second, and time zone minute elements of the timestamp
- the hour can be 24 if the minutes, seconds, and any fractional seconds are all zeroes.
- the separator character that follows the seconds element can be omitted if fractional seconds are not included.
- the number of digits of fractional seconds can vary from 0 to 12. An implicit specification of 0 is assumed if fractional seconds are omitted. The number of digits of fractional seconds determines the precision of the timestamp value.
- an optional single blank can be included between the time and the time zone elements.
- an optional time zone can be included, in one of the following formats:

– ±th:tm, with values ranging from -24:00 to +24:00. A value of -0:00 is treated the same as +0:00.
– ±th, with values ranging from -24 to +24 (an implicit specification of 00 is assumed for the time zone minute element)
– uppercase Z to specify UTC

The data type of the value depends on the content of the string constant (where *p* is the number of digits of fractional seconds in the constant):

- **TIMESTAMP(*p*) WITHOUT TIME ZONE** if the content of the string constant conforms to the rules in the Table 37 table.
- **TIMESTAMP(*p*) WITH TIME ZONE** if the content of the string constant conforms to the rules in the Table 38 table.

Table 37. Formats used to specify a value for a data type of TIMESTAMP WITHOUT TIME ZONE

Description	TIMESTAMP(0) WITHOUT TIME ZONE [13]	TIMESTAMP(*p*) WITHOUT TIME ZONE [23]
Blank between date and time portions and colons in time portion.	• yyyy-mm-dd hh:mm:ss • yyyy-mm-dd hh:mm:ss.	yyyy-mm-dd hh:mm:ss.nnnnnnnnnnnn
Minus sign between date and time portions and periods in time portion.	• yyyy-mm-dd-hh.mm.ss • yyyy-mm-dd-hh.mm.ss.	yyyy-mm-dd-hh.mm.ss.nnnnnnnnnnnn
Blank between date and time portions and periods in time portion.	• yyyy-mm-dd hh.mm.ss • yyyy-mm-dd hh.mm.ss.	yyyy-mm-dd hh.mm.ss.nnnnnnnnnnnn

Notes:

1. No fractional seconds; shown with and without optional trailing period after seconds
2. *p* is the number of digits of fractional seconds. *nnnnnnnnnnnn* can range from 1 to 12 instances of *n*
3. As an additional format, the character T can be substituted as the separator between the date and time portions of the value.

Table 38. Formats used to specify a value for a data type of TIMESTAMP WITH TIME ZONE

Description	TIMESTAMP(0) WITH TIME ZONE[13]	TIMESTAMP(*p*) WITH TIME ZONE [23]
Blank between date and time portions and colons in time portion, no space between time and time zone.	• yyyy-mm-dd hh:mm:ss±th:tm • yyyy-mm-dd hh:mm:ss±th • yyyy-mm-dd hh:mm:ss.±th:tm • yyyy-mm-dd hh:mm:ss.±th	• yyyy-mm-dd hh:mm:ss.nnnnnnnnnnnn ±th:tm • yyyy-mm-dd hh:mm:ss.nnnnnnnnnnnn ±th
Minus sign between date and time portions and periods in time portion.	• yyyy-mm-dd-hh.mm.ss±th:tm • yyyy-mm-dd-hh.mm.ss±th • yyyy-mm-dd-hh.mm.ss.±th:tm • yyyy-mm-dd-hh.mm.ss.±th	• yyyy-mm-dd-hh.mm.ss.nnnnnnnnnnnn ±th:tm • yyyy-mm-dd-hh.mm.ss.nnnnnnnnnnnn ±th

Table 38. Formats used to specify a value for a data type of TIMESTAMP WITH TIME ZONE (continued)

Description	TIMESTAMP(0) WITH TIME ZONE[13]	TIMESTAMP(*p*) WITH TIME ZONE [23]
Blank between date and time portions, colons in time portion, blank between fractional seconds and sign for time zone.	• yyyy-mm-dd hh:mm:ss ±th:tm • yyyy-mm-dd hh:mm:ss ±th • yyyy-mm-dd hh:mm:ss. ±th:tm • yyyy-mm-dd hh:mm:ss. ±th	• yyyy-mm-dd hh:mm:ss.nnnnnnnnnnnn ±th:tm • yyyy-mm-dd hh:mm:ss.nnnnnnnnnnnn ±th
Blank between date and time portions and periods in time portion.	• yyyy-mm-dd hh.mm.ss±th:tm • yyyy-mm-dd hh.mm.ss±th • yyyy-mm-dd hh.mm.ss.±th:tm • yyyy-mm-dd hh.mm.ss.±th	• yyyy-mm-dd hh.mm.ss.nnnnnnnnnnnn ±th:tm • yyyy-mm-dd hh.mm.ss.nnnnnnnnnnnn ±th

Notes:

1. No fractional seconds; shown with and without optional trailing period after seconds
2. *p* is the number of digits of fractional seconds. *nnnnnnnnnnnn* can range from 1 to 12 instances of *n*
3. As an additional format, the character T can be substituted as the separator between the date and time portions of the value.

Graphic string constants

A *graphic string constant* specifies a varying-length graphic string.

In EBCDIC environments, the forms of graphic string constants are shown in the following figure. (Shift-in and shift-out characters for EBCDIC data are discussed in "Character strings" on page 84.)[6]

6. The PL/I form of graphic string constants is supported only in static SQL statements.

Context	Graphic String Constant	Empty String	Example
PL/I	⍾'dbcs-string'G⍿ ⍾'dbcs-string'⍿ G G represents a DBCS G (X'42C7') ' represents a DBCS apostrophe (X'427D')	⍾''G⍿ ⍾''⍿ G	⍾'元気'G⍿
All other contexts	G'⍾dbcs-string⍿'	G'⍾⍿' G'' g'⍾⍿' g''	G'⍾ 元気⍿'
	N'⍾dbcs-string⍿'	N'⍾⍿' N'' n'⍾⍿' n''	

Figure 17. Graphic string constants in EBCDIC

In SQL statements and in host language statements in a source program, graphic string constants cannot be continued from one line to the next. A graphic string constant must be short enough so that its UTF-8 representation requires no more than 32704 bytes.

DB2 supports two types of hexadecimal graphic string constants.

- UX'xxxx' represents a string of graphic Unicode UTF-16 characters, where *x* is a hexadecimal digit. The number of digits must be a multiple of 4 and must not exceed 32704. Each group of 4 digits represents a single UTF-16 graphic character. For example, the UX constant for 'ABC' is UX'004100420043'.
- GX'xxxx' represents a string of graphic characters, where *x* is a hexadecimal digit. The number of digits must be a multiple of 4. Each group of 4 digits represents a single double-byte graphic character. The hexadecimal shift-in and shift-out ('OE'X and 'OF'X), which apply to EBCDIC only, are not included in the string.

  If the MIXED DATA installation option is set to NO, a GX constant cannot be used. Instead, a UX constant should be used. A GX constant cannot be used when the encoding scheme is UNICODE.

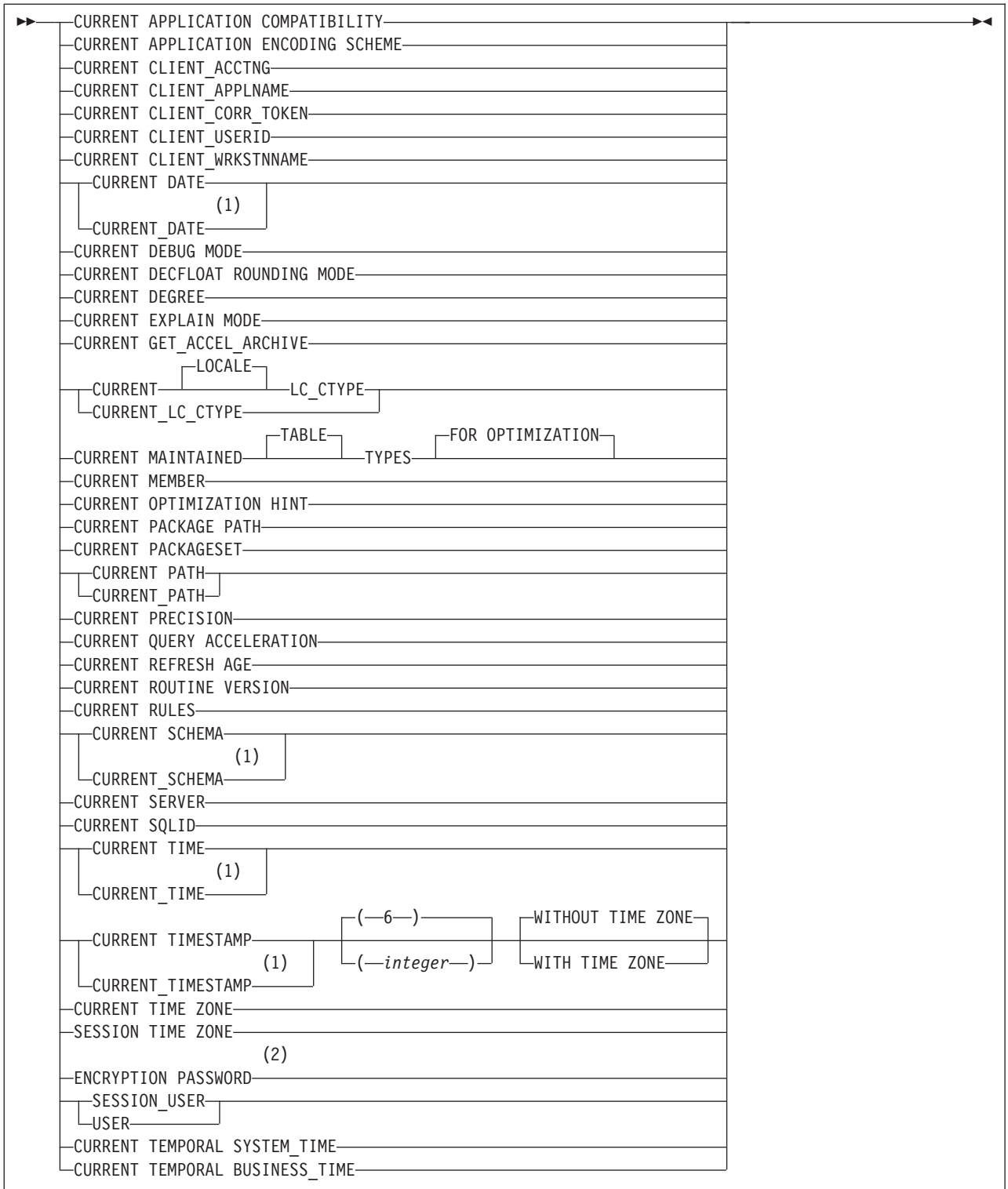
For information about the CCSID that is assigned to a graphic string constant, including UX'xxxx' and GX'xxxx' string constants, see “Determining the encoding scheme and CCSID of a string” on page 47.

Special registers

A special register is a storage area that is defined for an application process by DB2 and is used to store information that can be referenced in SQL statements. A reference to a special register is a reference to a value provided by the current server. If the value is a string, its CCSID is a default CCSID of the current server.

The special registers can be referenced as follows:

special registers



Notes:

1 The SQL standard uses the form with the underline.

2 The ENCRYPTION PASSWORD special register can only be explicitly referenced in the SET ENCRYPTION PASSWORD statement. It is used implicitly used by the encryption and decryption functions.

General rules for special registers

Changing register values

A commit operation might cause special registers to be re-initialized. Whether a special register is affected by a commit depends on whether the special register has been explicitly set within the application process. For example, assume that the PATH special register has not been explicitly set with a SET PATH statement in the application process. After a commit, the value of PATH is re-initialized. For information on the initialization of PATH, which can take the current value of CURRENT SQLID into consideration, see "CURRENT SQLID" on page 193.

A rollback operation has no effect on the values of special registers. Nor does any SQL statement, with the following exceptions:

- SQL SET statements can change the values of the following special registers:
  - CURRENT APPLICATION COMPATIBILITY
  - CURRENT APPLICATION ENCODING SCHEME
  - CURRENT DEBUG MODE
  - CURRENT DECFLOAT ROUNDING MODE
  - CURRENT DEGREE
  - CURRENT EXPLAIN MODE
  - CURRENT GET_ACCEL_ARCHIVE
  - CURRENT LOCALE LC_CTYPE
  - CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION
  - CURRENT OPTIMIZATION HINT
  - CURRENT PACKAGE PATH
  - CURRENT PACKAGESET
  - CURRENT PATH
  - CURRENT PRECISION
  - CURRENT QUERY ACCELERATION
  - CURRENT REFRESH AGE
  - CURRENT ROUTINE VERSION
  - CURRENT RULES
  - CURRENT SCHEMA
  - CURRENT SQLID[7]
  - CURRENT TEMPORAL BUSINESS_TIME
  - CURRENT TEMPORAL SYSTEM_TIME
  - ENCRYPTION PASSWORD
  - SESSION TIME ZONE
- SQL CONNECT statements can change the value of CURRENT SERVER.

7. If the SET CURRENT SQLID statement is executed in a stored procedure or user-defined function package that has a dynamic SQL behavior other than run behavior, the SET CURRENT SQLID statement does not affect the authorization ID that is used for dynamic SQL statements in the package. The dynamic SQL behavior determines the authorization ID. For more information, see DYNAMICRULES bind option (DB2 Commands).

Changing register values from IBM Data Server clients and drivers

In addition to using SQL SET statements, you can use the following IBM Data Server client and driver interfaces to change the values of most of the special registers that are listed under "Changing register values" on page 158:

- IBM Data Server Driver for JDBC and SQLJ method DB2DataSource.setSpecialRegisters
- For non-Java clients, the <specialregisters> subsection in the in db2dsdriver.cfg file

Use of these interfaces has the following restrictions:

- You cannot change the values of the following special registers:
  - CURRENT APPLICATION ENCODING SCHEME
  - CURRENT PACKAGESET
- The special register name must be in uppercase, and have no extraneous blanks. For example, the following strings are invalid:
  - CURRENT Application compatibility

    The special register name must be specified with all uppercase characters: CURRENT APPLICATION COMPATIBILITY.
  - CURRENT FUNCTION PATH

    The special register name must be specified with only one space between words: CURRENT FUNCTION PATH.
- For CURRENT REFRESH AGE, the value 99999999999999 is not supported. Use the value ANY instead.

Determining register values

You can use various statements to determine the value of a special register. For instance, a SELECT statement, a SET statement, the VALUES statement (if the statement is within a trigger action) will provide the value of a special register. The following examples find the value of the CURRENT PRECISION special register:

```
SELECT CURRENT PRECISION FROM SYSIBM.SYSDUMMY1;
SET :hv = CURRENT PRECISION
VALUES(CURRENT PRECISION)
```

CCSIDS for register values

Special registers that contain character strings have an associated CCSID. The particular CCSID depends on the context in which the special register is referenced. For more information, see "Determining the encoding scheme and CCSID of a string" on page 47.

Datetime special registers

The datetime registers are named CURRENT DATE, CURRENT TIME, and CURRENT TIMESTAMP. Datetime special registers are stored in an internal format. When two or more of these registers are implicitly or explicitly specified in a single SQL statement, they represent the same point in time. A datetime special register is implicitly specified when it is used to provide the default value of a datetime column.

If the SQL statement in which a datetime special register is used is in a user-defined function or stored procedure that is within the scope of a trigger, DB2 uses the timestamp for the triggering SQL statement to determine the special register value.

The values of these special registers are based on:

- The time-of-day clock of the processor for the server executing the SQL statement
- The TIMEZONE parameter for this processor. The TIMEZONE parameter is in SYS1.PARMLIB(CLOCKXX).

To evaluate the references when the statement is being executed, a single reading from the time-of-day clock is incremented by the number of hours, minutes, and seconds specified by the TIMEZONE parameter. The values derived from this are assumed to be the local date, time, or timestamp, where local means local to the DB2 that executes the statement. This assumption is correct if the clock is set to local time and the TIMEZONE parameter is zero or the clock is set to UTC (Coordinated Universal Time) and the TIMEZONE parameter gives the difference from UTC.

Because the datetime special registers and the CURRENT TIMEZONE special register depend on the parameter PARMTZ(SYS1.PARMLIB(CLOCKXX)), their values are affected if the local time at the server is changed by the z/OS system command SET CLOCK. The values of the CURRENT DATE and CURRENT TIMESTAMP special registers might be affected if the local date at the server is changed by the system command SET DATE[8].

Where special registers are processed

In distributed applications, CURRENT APPLICATION ENCODING SCHEME, CURRENT SERVER, and CURRENT PACKAGESET are processed locally. All other special registers are processed at the server.

8. Whether the SET DATE command affects these special registers depends on the system level and the program temporary fix (PTF) level of the system.

CURRENT APPLICATION COMPATIBILITY

CURRENT APPLICATION COMPATIBILITY specifies the compatibility level support for dynamic SQL.

The data type is VARCHAR(10).

The initial value of CURRENT APPLICATION COMPATIBILITY is determined by the value of the APPLCOMPAT bind parameter for the package. The initial value of CURRENT APPLICATION COMPATIBILITY in a user-defined function or stored procedure is inherited according to the rules in Table 40 on page 205. Set the value with the SET APPLICATION COMPATIBILITY statement.

The following values are supported:

V10R1 The dynamic SQL statements in the package have V10R1 compatibility behavior.

V11R1 The dynamic SQL statements in the package have V11R1 compatibility behavior. This value is only allowed in new-function mode.

Example: Set the host variable *CS* to the compatibility level.

```
EXEC SQL SET :CS = CURRENT APPLICATION COMPATIBILITY;
```

Related reference:

BIND and REBIND options (DB2 Commands)

"SET CURRENT APPLICATION COMPATIBILITY" on page 1882

APPL COMPAT LEVEL field (APPLCOMPAT subsystem parameter) (DB2 Installation and Migration)

CURRENT APPLICATION ENCODING SCHEME

CURRENT APPLICATION ENCODING SCHEME specifies which encoding scheme is to be used for dynamic statements. It allows an application to indicate the encoding scheme that is used to process data. This register is not supported in REXX applications or in stored procedures written in REXX

The initial value of CURRENT APPLICATION ENCODING SCHEME is determined by the value of the ENCODING bind option or the APPLICATION ENCODING SCHEME option of the CREATE PROCEDURE or ALTER PROCEDURE statement for native SQL procedures if the option is specified. If the option was not specified, the initial value is the value of field DEFAULT APPLICATION ENCODING SCHEME on installation panel DSNTIPF. You can changes the value of the register by executing the statement SET CURRENT APPLICATION ENCODING SCHEME.

The value contained in the special register is a character representation of a CCSID. Although you can use the values ASCII, EBCDIC, or UNICODE to set the special register, what is stored in the special register is a character representation of the numeric CCSID that corresponds to the value used in the SET CURRENT APPLICATION ENCODING SCHEME statement. The value ASCII, EBCDIC, or UNICODE is not stored. The CCSID_ENCODING scalar function can be used to get a value of ASCII, EBCDIC, or UNICODE from a numeric CCSID value.

The data type is CHAR(8). If necessary, the value is padded on the right with blanks so that its length is 8 bytes.

For stored procedures and user-defined functions, the initial value of the CURRENT APPLICATION ENCODING SCHEME special register is determined by the value of the ENCODING bind option for the package that is associated with the procedure or function or by the APPLICATION ENCODING SCHEME option of the CREATE PROCEDURE or ALTER PROCEDURE statement for a native SQL procedure. If the option was not specified, the initial value is the value of the field DEFAULT APPLICATION ENCODING SCHEME field on installation panel DSNTIPF.

For triggers, the initial value of the CURRENT APPLICATION ENCODING SCHEME special register is the value of field DEFAULT APPLICATION ENCODING SCHEME on installation panel DSNTIPF.

Example: The CURRENT APPLICATION ENCODING SCHEME special register can be used like any other special register:

```
EXEC SQL VALUES(CURRENT APPLICATION ENCODING SCHEME) INTO :HV1;
EXEC SQL INSERT INTO T1 VALUES (CURRENT APPLICATION ENCODING SCHEME);
EXEC SQL SET :HV1 = CURRENT APPLICATION ENCODING SCHEME;
EXEC SQL SELECT C1 FROM T1 WHERE C1 = CURRENT APPLICATION ENCODING SCHEME;
```

Related reference:

"SET CURRENT APPLICATION ENCODING SCHEME" on page 1883

CURRENT CLIENT_ACCTNG

CURRENT CLIENT_ACCTNG contains the value of the accounting string from the client information that is specified for the connection.

The data type is VARCHAR(255).

The value of the special register can be changed by using one of the following application programming interfaces (APIs):

- Set Client Information (sqleseti)
- SQLSetConnectAttr (ODBC)
- java.sql.Connection.setClientInfo (JDBC)
- The RRS DSNRLI SIGNON, AUTH SIGNON, CONTEXT SIGNON, or SET_CLIENT_ID function
- The WLM_SET_CLIENT_INFO stored procedure

The value for the accounting string will be obtained first from the accounting string that is set by the SET_CLIENT_ID function, AUTH SIGNON function, or the Set Client Information (sqleseti) API, or alternatively from the accounting token set by RRSAF if accounting string has not been set.

The application compatibility value of the package determines the length and blank padding of the CURRENT CLIENT_ACCTNG special register returned.

If one of these APIs is not used to set the value of the special register, an empty string is returned when the special register is referenced.

Example: Get the current value of the accounting string for this connection.

```
SET :ACCT_STRING = CURRENT CLIENT_ACCTNG
```

Related concepts:

RRSAF connection functions (DB2 Application programming and SQL)

Application compatibility of packages (DB2 Application programming and SQL)

Related tasks:

Providing extended client information to the data source with IBM Data Server Driver for JDBC and SQLJ-only methods (DB2 Application Programming for Java)

Related reference:

WLM_SET_CLIENT_INFO stored procedure (DB2 Application programming and SQL)

sqle_client_info data structure

CURRENT CLIENT_APPLNAME

CURRENT CLIENT_APPLNAME contains the value of the application name from the client information that is specified for the connection.

The data type is VARCHAR(255).

The default application name varies, depending on the connection:

- If the connection is from a remote application client driver, the default is the application name as supplied by the driver. Default values set by the IBM Data Server Driver for JDBC and SQLJ can be obtained from the DatabaseMetaData.getClientInfoProperties method.
- If the connection is from a remote DB2 11 for z/OS application, the default varies depending on which attachment facility is used:

  TSO attachment facility
  The default application name is one of the following cases:
  - The TSO logon user ID when the application runs in TSO foreground using TSO online applications like SPUFI.
  - The job name when the application runs in TSO background using TSO batch applications like DSNTEP2.

  RRS attachment facility interface
  The correlation ID that is provided at the call of the RRS DSNRLI SIGNON function.

  Call attachment facility
  The job name.

  CICS attachment facility
  The first 8 bytes of the correlation ID. In particular the correlation ID is a 12-byte string for a CICS transaction, where the first 8 bytes are used as the default application name.

  IMS Attachment facility
  An 8-byte string, the Program Specification Block (PSB) name, or the program name.

The value of the special register can be changed by using one of the following application programming interfaces (APIs):

- SQLE_CLIENT_INFO_APPLNAME (sqleseti)
- SQLSetConnectAttr (ODBC)
- java.sql.Connection.setClientInfo (JDBC)
- The RRS DSNRLI SIGNON, AUTH SIGNON, CONTEXT SIGNON, or SET_CLIENT_ID function
- The WLM_SET_CLIENT_INFO stored procedure

When the client application name is explicitly set, it overwrites the default application name described above and is used as the client application name.

The application compatibility value of the package determines the length and blank padding of the CURRENT CLIENT_APPLNAME special register returned.

If one of these APIs is not used to set the value of the special register, an empty string is returned when the special register is referenced.

Example: Select the departments that are allowed to use the application that is being used in this connection.

```
SELECT DEPT
  FROM DEPT_APPL_MAP
  WHERE APPL_NAME = CURRENT CLIENT_APPLNAME
```

Related concepts:

RRSAF connection functions (DB2 Application programming and SQL)

Application compatibility of packages (DB2 Application programming and SQL)

Related tasks:

Providing extended client information to the data source with IBM Data Server Driver for JDBC and SQLJ-only methods (DB2 Application Programming for Java)

Related reference:

WLM_SET_CLIENT_INFO stored procedure (DB2 Application programming and SQL)

sqle_client_info data structure

CURRENT CLIENT_CORR_TOKEN

CURRENT CLIENT_CORR_TOKEN contains the value of the client correlation token from the client information that is specified for the connection.

The data type is VARCHAR(255).

The value of the special register can be changed by using one of the following application programming interfaces (APIs):

- SQLE_CLIENT_INFO_PROGRAMID (sqleseti)
- java.sql.Connection.setClientInfo (JDBC)
- The RRS DSNRLI SIGNON, AUTH SIGNON, CONTEXT SIGNON, or SET_CLIENT_ID function

If one of these APIs is not used to set the value of the special register, the value defaults to the correlation identifier from the client driver such as an application identifier. If a correlation identifier is not provided by the client system, an LUWID (Logical Unit of Work ID) is generated which becomes the correlation token.

Example: Select the departments that are allowed to use the correlation token that is being used in this connection.

```
SELECT DEPT
  FROM DEPT_CORR_TOKEN_MAP
  WHERE CORR_TOKEN_NAME = CURRENT CLIENT_CORR_TOKEN
```

Related concepts:

RRSAF connection functions (DB2 Application programming and SQL)

Related tasks:

Providing extended client information to the data source with IBM Data Server Driver for JDBC and SQLJ-only methods (DB2 Application Programming for Java)

Related reference:

WLM_SET_CLIENT_INFO stored procedure (DB2 Application programming and SQL)

sqle_client_info data structure

CURRENT CLIENT_USERID

CURRENT CLIENT_USERID contains the value of the client user ID from the client information that is specified for the connection.

The default client user ID is the primary authorization ID used to establish the connection.

The data type is VARCHAR(255).

The value of the special register can be changed by using one of the following application programming interfaces (APIs):

- SQLE_CLIENT_INFO_USERID (sqleseti)
- SQLSetConnectAttr (ODBC)
- java.sql.Connection.setClientInfo (JDBC)
- The RRS DSNRLI SIGNON, AUTH SIGNON, CONTEXT SIGNON, or SET_CLIENT_ID function
- The WLM_SET_CLIENT_INFO stored procedure

When the client user ID is explicitly set, it overwrites the primary authorization id described above and is used as the client user ID.

If one of these APIs is not used to set the value of the special register, an empty string is returned when the special register is referenced.

If the value set by the API exceeds 128 bytes, it is truncated to 128 bytes.

The application compatibility value of the package determines the length and blank padding of the CURRENT CLIENT_USERID special register returned.

Example: Find out in which department the current client user ID works.

```
SELECT DEPT
  FROM DEPT_USERID_MAP
  WHERE USER_ID = CURRENT CLIENT_USERID
```

Related concepts:

RRSAF connection functions (DB2 Application programming and SQL)

Application compatibility of packages (DB2 Application programming and SQL)

Related tasks:

Providing extended client information to the data source with IBM Data Server Driver for JDBC and SQLJ-only methods (DB2 Application Programming for Java)

Related reference:

WLM_SET_CLIENT_INFO stored procedure (DB2 Application programming and SQL)

sqle_client_info data structure

Client info properties support by the IBM Data Server Driver for JDBC and SQLJ (DB2 Application Programming for Java)

CURRENT CLIENT_WRKSTNNAME

CURRENT CLIENT_WRKSTNNAME contains the value of the workstation name from the client information that is specified for the connection.

The data type is VARCHAR(255).

The default workstation name varies, depending on the connection:

- If the connection originates from a DB2 11 for z/OS requester, it is the client host name.
- If the connection is from a remote application client driver, it is the client host name where the request is submitted. Default values that are set by the IBM Data Server Driver for JDBC and SQLJ can be obtained from the DatabaseMetaData.getClientInfoProperties method.
- If the connection is from a remote DB2 11 for z/OS application, the default varies depending on which attachment facility is used:

  TSO attachment facility
  The default workstation name is one of the following cases:
  - The default workstation name is 'TSO' when the application runs in TSO foreground with TSO online applications like SPUFI.
  - The default workstation name is 'BATCH' when the application runs in TSO background with TSO batch applications like DSNTEP2.

  RRS attachment facility interface
  The default workstation name is 'RRSAF

  Call attachment facility
  The default workstation name is 'DB2CALL'.

  CICS attachment facility
  The default workstation name is the CICS region name.

  IMS Attachment facility
  The default workstation name is IMS region ID.

The value of the special register can be changed by using one of the following application programming interfaces (APIs):

- SQLE_CLIENT_INFO_WRKSTNNAME (sqleseti)
- SQLSetConnectAttr (ODBC)
- java.sql.Connection.setClientInfo (JDBC)
- The RRS DSNRLI SIGNON, AUTH SIGNON, CONTEXT SIGNON, or SET_CLIENT_ID function
- The WLM_SET_CLIENT_INFO stored procedure

When the client workstation name is explicitly set, it overwrites the default workstation name described above and is used as the client workstation name.

The application compatibility value of the package determines the length and blank padding of the CURRENT CLIENT_WRKSTNNAME special register returned.

If one of these APIs is not used to set the value of the special register, an empty string is returned when the special register is referenced.

Example: Get the name of the workstation that is being used in this connection.

```
SET :WS_NAME = CURRENT CLIENT_WRKSTNNAME
```

Related concepts:

RRSAF connection functions (DB2 Application programming and SQL)

Application compatibility of packages (DB2 Application programming and SQL)

Related tasks:

Providing extended client information to the data source with IBM Data Server Driver for JDBC and SQLJ-only methods (DB2 Application Programming for Java)

Related reference:

WLM_SET_CLIENT_INFO stored procedure (DB2 Application programming and SQL)

sqle_client_info data structure

Client info properties support by the IBM Data Server Driver for JDBC and SQLJ (DB2 Application Programming for Java)

CURRENT DATE

The CURRENT DATE special register specifies a date that is based on a reading of the time-of-day clock when the SQL statement is executed at the current server.

If this special register is used more than one time within a single SQL statement, or used with CURRENT TIME or CURRENT TIMESTAMP within a single statement, all values are based on a single clock reading.[9]

The value of CURRENT DATE in a user-defined function or stored procedure is inherited according to the rules in Table 40 on page 205. For other applications, the date is derived by the DB2 that executes the SQL statement that refers to the special register. For a description of how the date is derived, see Datetime special registers.

Specifying CURRENT_DATE is equivalent to specifying CURRENT DATE.

Example: Display the average age of employees.

```
SELECT AVG(YEAR(CURRENT DATE - BIRTHDATE))
  FROM DSN8B10.EMP;
```

9. Except for the case of a non-atomic multiple row INSERT or MERGE statement.

CURRENT DEBUG MODE

CURRENT DEBUG MODE specifies the default value for the DEBUG MODE option when certain routines are created. The DEBUG MODE option specifies whether the routine should be built with the ability to run in debugging mode.

CURRENT DEBUG MODE specifies the default value for the DEBUG MODE option of the following statements:

- ALTER FUNCTION for a new version of an SQL scalar function
- ALTER PROCEDURE for a new version of a native SQL procedure
- CREATE FUNCTION for an SQL scalar function
- CREATE PROCEDURE for a Java procedure
- CREATE PROCEDURE for a native SQL procedure

The data type is VARCHAR(8). The following values are valid:

- ALLOW — Specifies that the routine can be run in debugging mode.
- DISALLOW — Specifies that the routine cannot be run in debugging mode. A subsequent ALTER statement can change the DEBUG MODE option to allow the routine to run in debugging mode.
- DISABLE — Specifies that the routine can never be run in debugging mode. When DISABLE is in effect, the routine cannot be changed to run in debugging mode. A subsequent ALTER statement cannot change the DEBUG MODE option to allow or disallow the routine to run in debugging mode.

The value of CURRENT DEBUG MODE in a user-defined function or stored procedure is inherited according to the rules in Table 40 on page 205. In other contexts the initial value of CURRENT DEBUG MODE is DISALLOW.

You can change the value of the CURRENT DEBUG MODE special register by running the SET CURRENT DEBUG MODE statement.

Example: Set the host variable DEBUG_MODE_OPT to the value of the CURRENT DEBUG MODE special register:

```
VALUES CURRENT DEBUG MODE INTO :DEBUG_MODE_OPT;
```

Related reference:

"SET CURRENT DEBUG MODE" on page 1884

CURRENT DECFLOAT ROUNDING MODE

CURRENT DECFLOAT ROUNDING MODE specifies the default rounding mode that is used for DECFLOAT values.

The data type is VARCHAR(128). The following rounding modes are supported:

- ROUND_CEILING — rounds the value towards positive infinity. If all of the discarded digits are zero or if the sign is negative the result is unchanged other than the removal of the discarded digits. Otherwise, the result coefficient is incremented by 1.
- ROUND_DOWN — rounds the value towards 0 (truncation). The discarded digits are ignored.
- ROUND_FLOOR — rounds the value towards negative infinity. If all of the discarded digits are zero or if the sign is positive the result is unchanged other than the removal of discarded digits. Otherwise, the sign is negative and the result coefficient is incremented by 1.
- ROUND_HALF_DOWN — rounds the value to the nearest value; if the values are equidistant, rounds the value towards zero. If the discarded digits represent greater than half (0.5) of the value of a one in the next left position then the result coefficient is incremented by 1. Otherwise the discarded digits are ignored. This rounding mode is not recommended when creating a portable application because it is not supported by the IEEE draft standard for floating-point arithmetic.
- ROUND_HALF_EVEN — rounds the value to the nearest value; if the values are equidistant, rounds the value so that the final digit is even. If the discarded digits represents greater than half (0.5) of the value of one in the next left position then the result coefficient is incremented by 1. If they represent less than half, then the result coefficient is not adjusted (that is, the discarded digits are ignored). Otherwise the result coefficient is unaltered if its rightmost digit is even, or is incremented by 1 if its rightmost digit is odd (to make an even digit).
- ROUND_HALF_UP — rounds the value to the nearest value; if the values are equidistant, rounds the value away from zero. If the discarded digits represent greater than or equal to half (0.5) of the value of one in the next left position then the result coefficient is incremented by 1. Otherwise the discarded digits are ignored.
- ROUND_UP — rounds the value away from 0. If all of the discarded digits are zero the result is unchanged other than the removal of discarded digits. Otherwise, the result coefficient is incremented by 1. This rounding mode is not recommended when creating a portable application because it is not supported by the IEEE draft standard for floating-point arithmetic.

The initial value of CURRENT DECFLOAT ROUNDING MODE is the value of the ROUNDING bind option or the native SQL procedure option. If the ROUNDING option is not specified, the initial value is the value of the DEF DECFLOAT ROUND MODE field on installation panel DSNTIPF.

The value of CURRENT DECFLOAT ROUNDING MODE in a user-defined function or stored procedure is inherited according to the rules in Table 40 on page 205.

You can change the value of the CURRENT DECFLOAT ROUNDING MODE by executing the statement SET CURRENT DECFLOAT ROUNDING MODE.

Example: Set the DECFLOAT rounding mode to ROUND_CEILING:

```
SET CURRENT DECFLOAT ROUNDING MODE = 'ROUND_CEILING';
```

Related reference:
"SET CURRENT DECFLOAT ROUNDING MODE" on page 1886

CURRENT DEGREE

CURRENT DEGREE specifies the degree of parallelism for the execution of queries that are dynamically prepared by the application process.

The data type of the register is CHAR(3) and the only valid values are 1 (padded on the right with two blanks) and ANY.

If the value of CURRENT DEGREE is 1 when a query is dynamically prepared, the execution of that query will not use parallelism. If the value of CURRENT DEGREE is ANY when a query is dynamically prepared, the execution of that query can involve parallelism.

The initial value of CURRENT DEGREE is determined by the value of field CURRENT DEGREE on installation panel DSNTIP8. The default for the initial value of that field is 1 unless your installation has changed it to be ANY by modifying the value in that field. The initial value of CURRENT DEGREE in a user-defined function or stored procedure is inherited according to the rules in Table 40 on page 205.

You can change the value of the register by executing the statement SET CURRENT DEGREE.

CURRENT DEGREE is a register at the database server. Its value applies to queries that are dynamically prepared at that server and to queries that are dynamically prepared at another DB2 subsystem as a result of the use of a DB2 private connection between that server and that DB2 subsystem.

Example: The following statement inhibits parallelism:

```
SET CURRENT DEGREE = '1';
```

Related concepts:

Parallel processing (DB2 Performance)

Related tasks:

Enabling parallel processing (DB2 Performance)

Disabling query parallelism (DB2 Performance)

Related reference:

"SET CURRENT DEGREE" on page 1889

CURRENT DEGREE field (CDSSRDEF subsystem parameter) (DB2 Installation and Migration)

CURRENT EXPLAIN MODE

The CURRENT EXPLAIN MODE special register contains the values that control the behavior EXPLAIN in regard to eligible dynamic SQL statements.

This facility generates and inserts EXPLAIN information into the EXPLAIN tables. Possible values for the CURRENT EXPLAIN MODE special register are YES, NO, and EXPLAIN. The data type is VARCHAR(128).

NO Disable the ability to use EXPLAIN. No EXPLAIN information is kept. NO is the initial value of the EXPLAIN MODE special register.

YES
Enables the EXPLAIN facility and causes EXPLAIN information to be inserted into the EXPLAIN tables for eligible dynamic SQL statements after the statement is prepared and executed. All dynamic SQL statements are compiled and executed normally.

EXPLAIN
Enables the EXPLAIN facility and causes EXPLAIN information to be captured for any eligible dynamic SQL statement after the statement is prepared. This setting behaves similarly to YES, however, dynamic statements, except for SET statements, are not executed.

For values YES and EXPLAIN, prepared statements are not saved into the dynamic statement cache.

The initial value is NO. The initial value of CURRENT EXPLAIN MODE in a user-defined function or stored procedure is inherited according to the rules in Table 40 on page 205.

The value can be changed using the SET CURRENT EXPLAIN MODE statement.

Prerequisites for using CURRENT EXPLAIN MODE:

- Both the PLAN_TABLE and DSN_STATEMENT_CACHE_TABLE exist on the DB2 server and the table names are qualified with the current SQLID that is used when running the application.
- The Dynamic statement cache is enabled.
- The client application contains some explainable statements.

Required authorization for using CURRENT EXPLAIN MODE:
If CURRENT EXPLAIN MODE is set to YES or EXPLAIN, the privilege set for the underlying statement must have the necessary authorization to use the EXPLAIN facility.

When the EXPLAIN privilege is in effect and CURRENT EXPLAIN MODE is set to EXPLAIN, any SQLCODE that is returned due to the EXPLAIN privilege override any SQLCODE that is returned due to CURRENT EXPLAIN MODE being set to EXPLAIN.

Related reference:

"SET CURRENT EXPLAIN MODE" on page 1891

CURRENT GET_ACCEL_ARCHIVE

The CURRENT GET_ACCEL_ARCHIVE special register specifies whether a query that references a table that is archived in an accelerator server uses the archived data.

The data type is VARCHAR(255).

Valid values are:

NO Specifies that if a table is archived in an accelerator server, and a query references that table, the query does not use the data that is archived.

YES
Specifies that if a table is archived in an accelerator server, and a query references that table, the query uses the data that is archived.

The initial value of CURRENT GET_ACCEL_ARCHIVE is determined by the value of DB2 subsystem parameter GET_ACCEL_ARCHIVE. The default for the initial value of that subsystem parameter is NO unless your installation has changed the value. The initial value of CURRENT GET_ACCEL_ARCHIVE in a user-defined function or stored procedure is inherited according to the rules in Table 40 on page 205.

You can change the value of the register by executing the SET CURRENT GET_ACCEL_ARCHIVE statement.

Example: The following statement sets the CURRENT GET_ACCEL_ARCHIVE special register so that when a table is archived in an accelerator server, the table reference does not include the archived data.

```
SET CURRENT GET_ACCEL_ARCHIVE=NO;
```

Related reference:

"SET CURRENT GET_ACCEL_ARCHIVE" on page 1893

CURRENT LOCALE LC_CTYPE

CURRENT LOCALE LC_CTYPE specifies the LC_CTYPE locale that will be used to execute SQL statements that use a built-in function that references a locale. Functions LCASE, UCASE, and TRANSLATE (with a single argument) refer to the locale when they are executed.

The data type is CHAR(50). If necessary, the value is padded on the right with blanks so that its length is 50 bytes. The following values are supported:

- blank — For a conversion to lowercase, SBCS uppercase characters A-Z are converted to SBCS lowercase characters a-z, and characters with diacritical marks are not converted. If the string contains MIXED or DBCS characters, full-width Latin uppercase characters A-Z are converted to full-width lowercase characters a-z.

  For a conversion to uppercase, SBCS lowercase characters a-z are converted to SBCS uppercase characters A-Z, and characters with diacritical marks are not converted. If the string contains MIXED or DBCS characters, full-width Latin lowercase characters a-z are converted to full-width uppercase characters A-Z.

  For optimal performance, specify a blank string unless your data must be processed by using rules that are defined by a specific locale.
- UNI — Case conversions use both the NORMAL and SPECIAL casing capabilities as described in *z/OS Support for Unicode: Using Unicode Services*. UNI cannot be used with EBCDIC data.
- locale name — The locale defines the rules for conversion to uppercase or lowercase characters. For information on locales and their naming conventions for EBCDIC data, see *z/OS C/C++ Programming Guide*. For information on locales and their naming conventions for Unicode and ASCII data, see *z/OS Support for Unicode: Using Unicode Services*.

The initial value of CURRENT LOCALE LC_CTYPE is determined by the value of field LOCALE LC_CTYPE on installation panel DSNTIPF. The default for the initial value of that field is blank unless your installation has changed the value of that field. The initial value of CURRENT LOCALE LC_CTYPE in a user-defined function or stored procedure is inherited according to the rules in Table 40 on page 205.

You can change the value of the register by executing the statement SET CURRENT LOCALE LC_CTYPE.

Some examples of locales for EBCDIC data include:

Fr_BE
Fr_FR@EURO
En_US
Ja_JP

Example: Save the value of current register CURRENT LOCALE LC_CTYPE in host variable *HV1*, which is defined as VARCHAR(50).

```
EXEC SQL VALUES(CURRENT LOCALE LC_CTYPE) INTO :HV1;
```

Related concepts:

z/OS: Unicode Services User's Guide and Reference

Related reference:

"SET CURRENT LOCALE LC_CTYPE" on page 1894

z/OS XL C/C++ Programming Guide

CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION

CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION specifies a value that identifies the types of objects that can be considered to optimize the processing of dynamic SQL queries. This register contains a keyword representing table types.

The data type is VARCHAR(255).

The initial value of CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION is determined by the value of field CURRENT MAINT TYPES on installation panel DSNTIP81. The default for the initial value of that field is SYSTEM unless your installation has changed the value of that field. The initial value of CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION in a user-defined function or stored procedure is inherited according to the rules in Table 40 on page 205.

You can change the value of the register by executing the SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION statement. The object types controlled by this special register are never considered by static embedded SQL queries.

Example: Set the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register so that all materialized query tables are considered.

```
SET CURRENT MAINTAINED TABLE TYPES ALL;
```

Related reference:

"SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION" on page 1896

CURRENT MEMBER

CURRENT MEMBER specifies the member name of a current DB2 data sharing member on which a statement is executing. The value of CURRENT MEMBER is a character string.

The data type is CHAR(8). If necessary, the member name is padded to the right with blanks so that its length is 8 bytes.

The value of a CURRENT MEMBER is a string of blanks when the application process is connected to a DB2 subsystem that is not a member of a data sharing group.

The SQL SET statement cannot change the value of CURRENT MEMBER.

Example: Use one of the following statements to set the host variable *MEM* to the name of the current DB2 member.

```
EXEC SQL SET :MEM = CURRENT MEMBER;
EXEC VALUES (CURRENT MEMBER) into :MEM;
```

CURRENT OPTIMIZATION HINT

CURRENT OPTIMIZATION HINT specifies the user-defined optimization hint that DB2 should use to generate the access path for dynamic statements.

The data type is VARCHAR(128).

The value of the register identifies the rows in *owner*.PLAN_TABLE that DB2 uses to generate the access path. DB2 uses information in the rows in *owner*.PLAN_TABLE for which the value of the OPTHINT column matches the value of the CURRENT OPTIMIZATION special register. If the value of the register is an empty string or all blanks, DB2 uses normal optimization and ignores optimization hints. If the value of the register includes any non-blank characters and DB2 was installed without optimization hints enabled (field OPTIMIZATION HINTS on installation panel DSNTIP8), a warning occurs.

The initial value of CURRENT OPTIMIZATION HINT is the value of the OPTHINT bind option or of the native SQL procedure option. The initial value of CURRENT OPTIMIZATION HINT in a user-defined function or stored procedure is inherited according to the rules in Table 40 on page 205. You can change the value of the special register by executing the statement SET CURRENT OPTIMIZATION HINT.

Example: Set the CURRENT OPTIMIZATION HINT special register so that DB2 uses the optimization plan hint that is identified by host variable *NOHYB* when generating the access path for dynamic statements.

```
SET CURRENT OPTIMIZATION HINT = :NOHYB
```

Related tasks:

Specifying access paths in a PLAN_TABLE instance (DB2 Performance)

Preparing to influence access paths (DB2 Performance)

Related reference:

"SET CURRENT OPTIMIZATION HINT" on page 1898

CURRENT PACKAGE PATH

CURRENT PACKAGE PATH specifies a value that identifies the path used to resolve references to packages that are used to execute SQL statements. This special register applies to both static and dynamic statements.

The data type is VARCHAR(4096). The value can be an empty or blank string, or a list of one or more collection IDs, where the collection IDs are enclosed in double quotation marks and separated by commas. Any quotation marks within the string are repeated as they are in any delimited identifier. The delimiters and commas are included in the length of the special register.

The initial value of CURRENT PACKAGE PATH is an empty string. The value is a list of collections only if the application process has explicitly specified a list of collections by means of the SET CURRENT PACKAGE PATH statement.

The initial value of CURRENT PACKAGE PATH in a user-defined function or procedure is inherited according to the rules in Table 40 on page 205.

When CURRENT PACKAGE PATH or CURRENT PACKAGESET is set, DB2 uses the values in these registers to resolve the collection for a package. The value of CURRENT PACKAGE PATH takes priority over CURRENT PACKAGESET. In a distributed environment, the value of CURRENT PACKAGE PATH at the remote server takes precedence of the value of CURRENT PACKAGE PATH at the local server (the requester). For more information on package resolution, see *DB2 Application Programming and SQL Guide*.

Example: In an application that is using SQLJ packages (in collection SQLJ1 and SQLJ2) and a JDBC package in DB2JAVA, set the CURRENT PACKAGE PATH special register to check SQLJ1 first, followed by SQLJ2, and DB2JAVA:

```
SET CURRENT PACKAGE PATH = SQLJ1, SQLJ2, DB2JAVA;
```

The following statement sets the host variable to the value of the resulting list:

```
SET :HVPKLIST = CURRENT PACKAGE PATH;
```

The value of the host variable would be "SQLJ1", "SQLJ2", "DB2JAVA".

Related reference:

"SET CURRENT PACKAGE PATH" on page 1899

CURRENT PACKAGESET

CURRENT PACKAGESET specifies an empty string, a string of blanks, or the collection ID of the package that will be used to execute SQL statements.

The data type is VARCHAR(128).

The initial value of CURRENT PACKAGESET is an empty string. The value is a collection ID only if the application process has explicitly specified a collection ID by means of the SET CURRENT PACKAGESET statement.

The initial value of CURRENT PACKAGESET in a user-defined function or stored procedure is inherited according to the rules in Table 40 on page 205.

Example: Before passing control to another program, identify the collection ID for its package as ALPHA.

```
EXEC SQL SET CURRENT PACKAGESET = 'ALPHA';
```

Related reference:

"SET CURRENT PACKAGESET" on page 1903

CURRENT PATH

CURRENT PATH specifies the SQL path used to resolve unqualified data type names and function names in dynamically prepared SQL statements. It is also used to resolve unqualified procedure names that are specified as host variables in SQL CALL statements (CALL *host-variable*).

The data type is VARCHAR(2048).

The CURRENT PATH special register contains a list of one or more schema names, where each schema name is enclosed in delimiters and separated from the following schema by a comma (any delimiters within the string are repeated as they are in any delimited identifier). The delimiters and commas are included in the 2048 character length.

For information on when the SQL path is used to resolve unqualified names in both dynamic and static SQL statements and the effect of its value, see "SQL path" on page 64.

The initial value of the CURRENT PATH special register is either:

- The value of the PATH bind option
- The SQL PATH option of the CREATE PROCEDURE or ALTER PROCEDURE statement for native SQL procedures
- "SYSIBM", "SYSFUN", "SYSPROC", "SYSIBMADM", "*value of CURRENT SQLID special register*" if the PATH bind option or SQL PATH option was not specified
- "SYSIBM", "SYSFUN", "SYSPROC", "SYSIBMADM", "*value of the role name that is associated with the user in the trusted context*" if the PATH bind option or SQL PATH option was not specified and if the connection is trusted with the role as object owner and qualifier options are in effect.

If the value of the CURRENT SQLID special register changes after the initial value of PATH special register is established, the value of the PATH special register is unaffected when the CURRENT SQLID is updated. However, if a commit later occurs and a SET PATH statement has not been processed, the value of PATH special register is reinitialized taking into consideration the current value of the CURRENT SQLID special register.

The initial value of CURRENT PATH in a user-defined function or stored procedure is inherited according to the rules in Table 40 on page 205.

You can change the value of the register by executing the statement SET PATH. For portability across the platforms, it is recommended that a SET PATH statement be issued at the beginning of an application.

Example: Set the special register so that schema SMITH is searched before the system schemas:

```
SET PATH = SMITH, SYSTEM PATH;
```

Related reference:

"SET PATH" on page 1921

CURRENT PRECISION

CURRENT PRECISION specifies the rules to be used when both operands in a decimal operation have precisions of 15 or less.

The data type of the register is CHAR(5).

Valid values for the CURRENT PRECISION special register include 'DEC15', 'DEC31', or 'D*pp*.*s*' where '*pp*' is either 15 or 31 and '*s*' is a number between 1 and 9. DEC15 specifies the rules that do not allow a precision greater than 15 digits, and DEC31 specifies the rules that allow a precision of up to 31 digits. The rules for DEC31 are always used if either operand has a precision greater than 15. If the form 'D*pp*.*s*' is used, '*pp*' represents the precision that will be used as the rules where DEC15 and DEC31 rules are used, and '*s*' represents the minimum divide scale to use for division operations. The separator used in the form 'D*pp*.*s*' can be either the '.' or the ',' character, regardless of the setting of the default decimal point.

The initial value of CURRENT PRECISION is determined by the value of field DECIMAL ARITHMETIC on installation panel DSNTIP4. The default for the initial value is DEC15 unless your installation has changed it to be DEC31 by modifying the value in that field. The initial value of CURRENT PRECISION in a user-defined function or stored procedure is inherited according to the rules in Table 40 on page 205.

You can change the value of the register by executing the statement SET CURRENT PRECISION.

CURRENT PRECISION only affects dynamic SQL. When an SQL statement is dynamically prepared and the value of CURRENT PRECISION is DEC15 or D15.*s*, where '*s*' is a number between 1 and 9, DEC15 rules will apply. When an SQL statement is dynamically prepared and the value of CURRENT PRECISION is DEC31 or D31.*s*, where '*s*' is a number between 1 and 9, DEC31 rules will apply. Preparation of a statement with DEC31 instead of DEC15 is more likely to result in an error, especially for division operations. Specification of CURRENT PRECISION in the form 'D*pp*.*s*' where '*pp*' is either 15 or 31 and '*s*' represents the minimum divide scale, will in some cases make division errors less likely when '*pp*' is set to 31. For more information, see “Arithmetic with two decimal operands” on page 244.

Example 1: Set CURRENT PRECISION so that subsequent statements that are prepared use DEC31 rules for decimal arithmetic:

```
SET CURRENT PRECISION = 'DEC31';
```

Example 2: Set CURRENT PRECISION so that subsequent statements that are prepared use DEC31 rules for decimal arithmetic with a minimum divide scale of 3:

```
SET CURRENT PRECISION = 'D31.3';
```

Related reference:

“SET CURRENT PRECISION” on page 1905

CURRENT QUERY ACCELERATION

The CURRENT QUERY ACCELERATION special register specifies a value that identifies when DB2 sends queries to an accelerator server and what DB2 does if the accelerator server fails.

The data type is VARCHAR(255).

Valid values are:

NONE
Specifies that no queries are sent to an accelerator server.

ENABLE
Specifies that queries are accelerated only if DB2 determines that it is advantageous to do so. If an accelerator failure occurs while a query is running or if the accelerator returns an error, DB2 returns a negative SQLCODE to the application.

ENABLE WITH FAILBACK
Specifies that queries are accelerated only if DB2 determines that it is advantageous to do so. If the accelerator returns an error during the PREPARE or first OPEN for the query, DB2 executes the query without the accelerator. If the accelerator returns an error during a FETCH or a subsequent OPEN, DB2 returns the error to the user and does not execute the query.

ELIGIBLE
Specifies that queries are accelerated if they are eligible for acceleration. DB2 does not use cost information to determine whether to accelerate the queries. Queries that are not eligible for acceleration are executed by DB2. If an accelerator failure occurs while a query is running or if the accelerator returns an error, DB2 returns a negative SQLCODE to the application.

ALL
Specifies that queries are accelerated if they are eligible for acceleration. DB2 does not use cost information to determine whether to accelerate the queries. Queries that are not eligible for acceleration are not executed by DB2, and an SQL error is returned. If an accelerator failure occurs while a query is running or if the accelerator returns an error, DB2 returns a negative SQLCODE to the application.

The initial value of CURRENT QUERY ACCELERATION is determined by the value of DB2 subsystem parameter QUERY_ACCELERATION. The default for the initial value of that subsystem parameter is NONE unless your installation has changed the value. The initial value of CURRENT QUERY ACCELERATION in a user-defined function or stored procedure is inherited according to the rules in Table 40 on page 205.

You can change the value of the register by executing the SET CURRENT QUERY ACCELERATION statement.

Example: The following statement sets the CURRENT QUERY ACCELERATION special register so that no query acceleration occurs.

```
SET CURRENT QUERY ACCELERATION NONE;
```

Related reference:

"SET CURRENT QUERY ACCELERATION" on page 1906

CURRENT REFRESH AGE

CURRENT REFRESH AGE specifies a timestamp duration value. This duration is the maximum duration since a REFRESH TABLE statement has been processed on a system-maintained REFRESH DEFERRED materialized query table such that the materialized query table can be used to optimize the processing of a query. This special register affects dynamic statement cache matching.

The data type of the register is DECIMAL(20,6). For a description of durations, see "Datetime operands and durations" on page 254.

If CURRENT REFRESH AGE has a value of 99999999999999 (ANY), REFRESH DEFERRED materialized query tables are considered to optimize the processing of a dynamic SQL query. This value represents 9999 years, 99 months, 99 days, 99 hours, and 99 seconds.

The initial value of CURRENT REFRESH AGE is determined by the value of field CURRENT REFRESH AGE on installation panel DSNTIP81. The default for the initial value of that field is 0 unless your installation has changed it to ANY by modifying the value of that field. The initial value of CURRENT REFRESH AGE in a user-defined function or stored procedure is inherited according to the rules in Table 40 on page 205.

You can change the value of the register by executing the SET CURRENT REFRESH AGE statement.

Example : The following example retrieves the current value of the CURRENT REFRESH AGE special register into the host variable, *CURMAXAGE*:

```
EXEC SQL VALUES (CURRENT REFRESH AGE) INTO :CURMAXAGE;
```

The value would be '99999999999999.000000'.

Related reference:

"SET CURRENT REFRESH AGE" on page 1908

CURRENT ROUTINE VERSION

CURRENT ROUTINE VERSION specifies the version identifier that is to be used when invoking a native SQL procedure. CURRENT ROUTINE VERSION is used for CALL statements that use a host variable to specify the procedure name.

The data type of CURRENT ROUTINE VERSION is VARCHAR(64).

The initial value of CURRENT ROUTINE VERSION in a user-defined function or stored procedure is inherited according to the rules in Table 40 on page 205. In other contexts the initial value of CURRENT ROUTINE VERSION is an empty string. An empty string indicates that a version identifier is not in effect for the SQL routine. When an SQL routine that does not have a version identifier in effect is invoked, the currently active version (as indicated in the catalog) of that routine is used.

You can change the value of the CURRENT ROUTINE VERSION by executing the statement SET CURRENT ROUTINE VERSION.

Setting the CURRENT ROUTINE VERSION special register to a version identifier might affect native SQL procedures that are invoked until the value of CURRENT ROUTINE VERSION is changed. If a version of an SQL procedure has a version identifier that matches the version identifier in the special register, that version of the SQL procedure is used when the SQL procedure is invoked. If an SQL procedure does not have a version identifier that matches the version identifier in the special register, the currently active version of the SQL procedure (as defined in the catalog) is used when the SQL procedure is invoked.

Example: Set the host variable *ROUTINE_VER* to the value of the CURRENT ROUTINE VERSION special register:

```
VALUES CURRENT ROUTINE VERSION INTO :ROUTINE_VER;
```

Related reference:

"SET CURRENT ROUTINE VERSION" on page 1910

CURRENT RULES

CURRENT RULES specifies whether certain SQL statements are executed in accordance with DB2 rules or the rules of the SQL standard.

The data type of the register is CHAR(3), and the only valid values are 'DB2' and 'STD'.

CURRENT RULES is a register at the database server. If the server is not the local DB2, the initial value of the register is 'DB2'. Otherwise, the initial value is the same as the value of the SQLRULES bind option. The initial value of CURRENT RULES in a user-defined function or stored procedure is inherited according to the rules in Table 40 on page 205.

You can change the value of the register by executing the statement SET CURRENT RULES.

CURRENT RULES affects the statements listed in the following table. The table summarizes when the statements are affected and shows where to find detailed information. CURRENT RULES also affects whether DB2 issues an existence error (SQLCODE -204) or an authorization error (SQLCODE -551) when an object does not exist. For CURRENT RULES 'STD', DB2 issues an authorization error (SQLCODE -551) when an object does not exist instead of the existence error (SQLCODE -204).

Table 39. Summary of statements affected by CURRENT RULES

Statement	What is affected	Details in topic
ALTER TABLE	Enforcement of check constraints added. Default value of the delete rule for referential constraints. Whether DB2 creates LOB table spaces, auxiliary tables, and indexes on auxiliary tables for added LOB columns. Whether DB2 creates an index for an added ROWID column that is defined with GENERATED BY DEFAULT.	"ALTER TABLE" on page 984
CREATE TABLE	Default value of the delete rule for referential constraints. Whether DB2 creates LOB table spaces, auxiliary tables, and indexes on auxiliary tables for LOB columns if the table is explicitly created. Whether DB2 creates an index for a ROWID column that is defined with GENERATED BY DEFAULT if the table is explicitly created.	"CREATE TABLE" on page 1388
GRANT	Granting privileges to yourself.	"GRANT" on page 1695
REVOKE	Revoking privileges from authorization IDs	"REVOKE" on page 1812

Example: Set CURRENT RULES so that a later ALTER TABLE statement is executed in accordance with the rules of the SQL standard:

```
SET CURRENT RULES = 'STD';
```

Related reference:

"SET CURRENT RULES" on page 1912

CURRENT SCHEMA

The CURRENT SCHEMA special register specifies the schema name used to qualify unqualified database object references in dynamically prepared SQL statements.

The data type is VARCHAR(128).

For information on when the CURRENT SCHEMA is used to resolve unqualified names in dynamic SQL statements and the effect of its value, see "Qualification of unqualified object names" on page 65.

The CURRENT SCHEMA special register contains a value that is a single identifier without delimiters.

The initial value of the special register is the value of CURRENT SQLID at the time the connection is established. If the connection is established as a trusted connection with a role as the object owner and qualifier, the initial value of the special register is the value of the role name that is associated with the user in the trusted context. The initial value of the special register in a user-defined function or procedure is inherited according to the rules in Table 40 on page 205.

The value of the special register can be changed by executing the SET SCHEMA statement. The value of CURRENT SCHEMA is the same as the value of CURRENT SQLID unless a SET SCHEMA statement has been issued specifying a different value. After a SET SCHEMA statement has been issued in an application, the values of CURRENT SCHEMA and CURRENT SQLID are separate. Therefore, if the value of CURRENT SCHEMA needs to be changed, a SET SCHEMA statement must be issued.

Specifying CURRENT_SCHEMA is equivalent to specifying CURRENT SCHEMA.

Example: Set the schema for object qualification to 'D123'.

```
SET SCHEMA = 'D123'
```

CURRENT SERVER

CURRENT SERVER specifies the location name of the current server.

The data type is CHAR(16). If necessary, the location name is padded on the right with blanks so that its length is 16 bytes.

The initial value of CURRENT SERVER depends on the CURRENTSERVER bind option. If CURRENTSERVER *X* is specified on the bind subcommand, the initial value is *X*. If the option is not specified, the initial value is the location name of the local DB2. The initial value of CURRENT SERVER in a user-defined function or stored procedure is inherited according to the rules in Table 40 on page 205. The value of CURRENT SERVER is changed by the successful execution of a CONNECT statement.

The value of CURRENT SERVER is a string of blanks when either of the following conditions apply:

- The application process is in the unconnected state
- The application process is connected to a local DB2 subsystem that does not have a location name.

Example: Set the host variable *CS* to the location name of the current server.

```
EXEC SQL SET :CS = CURRENT SERVER;
```

CURRENT SQLID

CURRENT SQLID specifies the SQL authorization ID of the process.

The data type is VARCHAR(128).

The SQL authorization ID is:

- The authorization ID used for authorization checking on dynamically prepared CREATE, GRANT, and REVOKE SQL statements.
- The owner of a table space, database, storage group, or synonym created by a dynamically issued CREATE statement.

The initial value of CURRENT SQLID can be provided by the connection or sign-on exit routine. If not, the initial value is the primary authorization ID of the process. The value remains in effect until one of the following events occurs:

- The SQL authorization ID is changed by the execution of a SET CURRENT SQLID statement.
- A SIGNON or re-SIGNON request is received from a CICS transaction subtask or an IMS independent region.
- The DB2 connection is ended.
- When running in a trusted connection, the user is switched.

The initial value of CURRENT SQLID in a user-defined function or stored procedure is inherited according to the rules in Table 40 on page 205.

CURRENT SQLID can only be referred to in an SQL statement that is executed by the current server.

CURRENT SQLID cannot be a role.

Example: Set the SQL authorization ID to 'GROUP34' (one of the authorization IDs of the process).

```
SET CURRENT SQLID = 'GROUP34';
```

CURRENT TEMPORAL BUSINESS_TIME

Examples

When a query references an application-period temporal table and the value of the CURRENT TEMPORAL BUSINESS_TIME special register is not the null value, the query is affected as follows:

- If the columns of a BUSINESS_TIME period are defined as TIMESTAMP, the following period specification is implicit:

  ```
  FOR BUSINESS_TIME AS OF CURRENT TEMPORAL BUSINESS_TIME
  ```

- If the columns of a BUSINESS_TIME period are defined as DATE, the following period specification is implicit: :

  ```
  FOR BUSINESS_TIME AS OF CAST(CURRENT TEMPORAL BUSINESS_TIME AS DATE)
  ```

The initial value of the special register depends on the context as follows:

- If the special register is in a trigger, the initial value is inherited from the invoking application.
- If the special register is in a user-defined function or procedure that is defined with the INHERIT SPECIAL REGISTERS option, the initial value is inherited from the invoking application.
- If the special register is in a user-defined function or procedure that is defined with the DEFAULT SPECIAL REGISTERS option, the initial value is the null value.
- In other contexts, the initial value of the special register is the null value.

You can change the value of the special register by using the SET CURRENT TEMPORAL BUSINESS_TIME statement. If you change the value within a routine, that new value is not passed back to the invoking application.

Example of a query that references an application-period temporal table

Assume the following conditions:

- ATT is an application-period temporal table and POLICY_ID is a column in ATT.
- The value of the BUSTIMESENSITIVE bind option is YES.
- The value of CURRENT TEMPORAL BUSINESS_TIME is not null.

Then, suppose that you issue the following query:

```
SELECT * FROM ATT
WHERE POLICY_ID = 123;
```

DB2 generates an implicit BUSINESS_TIME period specification for the query as follows:

```
SELECT * FROM ATT
FOR BUSINESS_TIME AS OF CURRENT TEMPORAL BUSINESS_TIME
WHERE POLICY_ID = 123;
```

Example of a procedure that uses CURRENT TEMPORAL BUSINESS_TIME

Suppose that procedure MYPROC is defined as follows:

```
CREATE PROCEDURE MYPROC(OUT VAR1 VARCHAR(40), OUT VAR2 VARCHAR(40))
BEGIN
 SELECT CURRENT TEMPORAL BUSINESS_TIME INTO VAR1
 FROM SYSIBM.SYSDUMMY1;

 SET CURRENT TEMPORAL BUSINESS_TIME = TIMESTAMP('2011-01-01') + 5 DAYS ;
```

```
 SELECT CURRENT TEMPORAL BUSINESS_TIME INTO VAR2
 FROM SYSIBM.SYSDUMMY1;
END!
```

Suppose that the application defines string variables VAR1, VAR2, and VAR3 and contains the following SQL statements:

```
SET CURRENT TEMPORAL BUSINESS_TIME = TIMESTAMP('2008-01-01') + 5 DAYS ;

CALL MYPROC(VAR1, VAR2);

SELECT CURRENT TEMPORAL BUSINESS_TIME INTO VAR3
FROM SYSIBM.SYSDUMMY1;
```

After the execution of the SQL statements, the variables have the following values:

- VAR1 has value '2008-01-06-00.00.00.000000000000', which is the CURRENT TEMPORAL BUSINESS_TIME value that is set before the CALL statement invoked the procedure.
- VAR2 has value '2011-01-06-00.00.00.000000000000', which is the CURRENT TEMPORAL BUSINESS_TIME value that is set during the CALL statement.
- VAR3 has value '2008-01-06-00.00.00.000000000000', which is the CURRENT TEMPORAL BUSINESS_TIME value that is set before the CALL statement. The changes of the register value inside the procedure have no affect on the invoking application.

Example of a query that references CURRENT TEMPORAL BUSINESS_TIME

Assume that IN_TRAY is an application-period temporal table that contains users and subject lines for notes in the inbox. The following query returns the user ID and subject line for notes in the IN_TRAY table that were sent on the date that the CURRENT TEMPORAL BUSINESS_TIME special register is set to.

```
SELECT SOURCE, SUBJECT
FROM IN_TRAY
WHERE DATE (CURRENT TEMPORAL BUSINESS_TIME) = DATE (RECEIVED)
```

Related tasks:

Querying temporal tables (DB2 Administration Guide)

Related reference:

"table-reference" on page 773

BIND and REBIND options (DB2 Commands)

"SET CURRENT TEMPORAL BUSINESS_TIME" on page 1915

"Special registers in a user-defined function or a stored procedure" on page 205

CURRENT TEMPORAL SYSTEM_TIME

The CURRENT TEMPORAL SYSTEM_TIME special register specifies a TIMESTAMP(12) value that is used in the default SYSTEM_TIME period specification for references to system-period temporal tables.

Examples

When a query references a system-period temporal table and the value of the CURRENT TEMPORAL SYSTEM_TIME special register is not the null value, the following period specification is implicit:

```
FOR SYSTEM_TIME AS OF CURRENT TEMPORAL SYSTEM_TIME
```

The initial value of the special register depends on the context as follows:

- If the special register is in a trigger, the initial value is inherited from the invoking application.
- If the special register is in a user-defined function or procedure that is defined with the INHERIT SPECIAL REGISTERS option, the initial value is inherited from the invoking application.
- If the special register is in a user-defined function or procedure that is defined with the DEFAULT SPECIAL REGISTERS option, the initial value is the null value.
- In other contexts, the initial value of the special register is the null value.

You can change the value of the special register by using the SET CURRENT TEMPORAL SYSTEM_TIME statement. If you change the value within a routine, that new value is not passed back to the invoking application.

Example of a query that references a system-period temporal table

Assume the following conditions:

- STT is a system-period temporal table, and POLICY_ID is a column of STT.
- The value of the SYSTIMESENSITIVE bind option is YES.
- The value of CURRENT TEMPORAL SYSTEM_TIME is not null.

Then, suppose that you issue the following query:

```
SELECT * FROM STT
WHERE POLICY_ID = 123;
```

DB2 generates an implicit SYSTEM_TIME period specification for the query as follows:

```
SELECT * FROM STT
FOR SYSTEM_TIME AS OF CURRENT TEMPORAL SYSTEM_TIME
WHERE POLICY_ID = 123;
```

Example of a procedure that uses CURRENT TEMPORAL SYSTEM_TIME

Suppose that procedure MYPROC is defined as follows:

```
CREATE PROCEDURE MYPROC(OUT VAR1 VARCHAR(40), OUT VAR2 VARCHAR(40))
BEGIN
 SELECT CURRENT TEMPORAL SYSTEM_TIME INTO VAR1
 FROM SYSIBM.SYSDUMMY1;

 SET CURRENT TEMPORAL SYSTEM_TIME = TIMESTAMP('2011-01-01') + 5 DAYS ;

 SELECT CURRENT TEMPORAL SYSTEM_TIME INTO VAR2
 FROM SYSIBM.SYSDUMMY1;
END!
```

Suppose that the application defines string variables VAR1, VAR2, and VAR3 and contains the following SQL statements:

```
SET CURRENT TEMPORAL SYSTEM_TIME = TIMESTAMP('2008-01-01') + 5 DAYS ;

CALL MYPROC(VAR1, VAR2);

SELECT CURRENT TEMPORAL SYSTEM_TIME INTO VAR3
FROM SYSIBM.SYSDUMMY1
```

After the execution of the SQL statements, the variables have the following values:

- VAR1 has value '2008-01-06-00.00.00.000000000000', which is the CURRENT TEMPORAL SYSTEM_TIME value that is set before the CALL statement invoked the procedure.
- VAR2 has value '2011-01-06-00.00.00.000000000000', which is the CURRENT TEMPORAL SYSTEM_TIME value that is set during the CALL statement.
- VAR3 has value '2008-01-06-00.00.00.000000000000', which is the CURRENT TEMPORAL SYSTEM_TIME value that is set before the CALL statement. The changes of the register value inside the procedure have no affect on the invoking application.

Example of a query that references a system-period temporal table

Assume that IN_TRAY is a system-period temporal table that contains users and subject lines for notes in the inbox. The following query returns the user IDs and subject lines based on the state of the messages in IN_TRAY as of the date that is specified by the CURRENT TEMPORAL SYSTEM_TIME special register.

```
SELECT SOURCE, SUBJECT
FROM IN_TRAY
```

If the special register is set to a non-null value, the previous statement is equivalent to the following statement:

```
SELECT SOURCE, SUBJECT
FROM IN_TRAY
FOR SYSTEM_TIME AS OF CURRENT TEMPORAL SYSTEM_TIME
```

Related tasks:

Querying temporal tables (DB2 Administration Guide)

Related reference:

"table-reference" on page 773

BIND and REBIND options (DB2 Commands)

"SET CURRENT TEMPORAL SYSTEM_TIME" on page 1917

"Special registers in a user-defined function or a stored procedure" on page 205

CURRENT TIME

The CURRENT TIME special register specifies a time that is based on a reading of the time-of-day clock when the SQL statement is executed at the current server.

If this special register is used more than one time within a single SQL statement, or used with CURRENT DATE or CURRENT TIMESTAMP within a single statement, all values are based on a single clock reading.[10]

The value of CURRENT TIME in a user-defined function or stored procedure is inherited according to the rules in Table 40 on page 205. For other applications, the time is derived by the DB2 that executes the SQL statement that refers to the special register. For a description of how the date is derived, see Datetime special registers.

Specifying CURRENT_TIME is equivalent to specifying CURRENT TIME.

Example: Display information about all project activities and include the current date and time in each row of the result.

```
SELECT DSN8B10.PROJACT.*, CURRENT DATE, CURRENT TIME
  FROM DSN8B10.PROJACT;
```

10. Except for the case of a non-atomic multiple row INSERT or MERGE statement.

CURRENT TIMESTAMP

The CURRENT TIMESTAMP special register specifies a timestamp that is based on a reading of the time-of-day clock when the SQL statement is executed at the current server.

If this special register is used more than one time within a single SQL statement, or used with CURRENT DATE or CURRENT TIME within a single statement, all values are based on a single clock reading.[11]

The value of CURRENT TIMESTAMP in a user-defined function or stored procedure is inherited according to the rules in Table 40 on page 205.

Specifying CURRENT_TIMESTAMP is equivalent to specifying CURRENT TIMESTAMP.

If you want a timestamp with a specified precision, the special register can be referenced as CURRENT TIMESTAMP(*integer*), where *integer* can range from 0 to 12. The default precision is 6. SYSDATE can also be specified as a synonym for CURRENT TIMESTAMP(0).

If you want a timestamp with a time zone, the special register can be referenced as CURRENT TIMESTAMP (*integer*) WITH TIME ZONE, or CURRENT TIMESTAMP WITH TIME ZONE. SYSTIMESTAMP can be specified as an alternative to CURRENT TIMESTAMP(12) WITH TIME ZONE. The time zone is determined from the CURRENT TIME ZONE special register.

Note: If the CURRENT TIMESTAMP special register is referenced in a timestamp with time zone context (for example, when compared with a timestamp with time zone column) the implicit time zone for the CURRENT TIMESTAMP special register will be based on the implicit time zone system parameter, which could be a different value from the CURRENT TIME ZONE special register. To avoid misinterpretation of the time zone in this case, CURRENT TIMESTAMP WITH TIME ZONE should be used.

Example 1: Display information about the full image copies that were taken in the last week.

```
SELECT * FROM SYSIBM.SYSCOPY
  WHERE TIMESTAMP > CURRENT TIMESTAMP - 7 DAYS;
```

Example 2: Insert a row into the IN_TRAY table. The value of the RECEIVED column should be a timestamp that indicates when the row was inserted. The values for the other three columns come from the host variables *SRC* (CHAR(8)), *SUB* (CHAR(64)), and *TXT* (VARCHAR(200)).

```
INSERT INTO IN_TRAY
 VALUES (CURRENT TIMESTAMP, :SRC, :SUB, :TXT)
```

Example 3: Retrieve the value of the CURRENT TIMESTAMP special register with a precision of 8 and include the time zone:

```
SELECT CURRENT TIMESTAMP(8) WITH TIME ZONE
 FROM SYSIBM.SYSDUMMY1;
```

11. Except for the case of a non-atomic multiple row INSERT or MERGE statement.

CURRENT TIME ZONE

The CURRENT TIME ZONE special register specifies a value that contains the difference between UTC and local time at the current server, if the SESSION TIME ZONE special register has not been set.

The data type is DECIMAL(6,0).

The difference between UTC and local time at the current server is represented by a time duration. A time duration is a decimal number in which the first two digits are the number of hours, the next two digits are the number of minutes, and the last two digits are the number of seconds. The number of hours is adjusted, if necessary, to fit in the range between -24 and 24 exclusive.

Subtracting CURRENT TIME ZONE from a local time converts that local time to UTC.

CURRENT TIMEZONE can be specified as an alternative to CURRENT TIME ZONE.

Example: Select all the rows of the IN_TRAY table. Assume that the RECEIVED column is defined as TIMESTAMP WITHOUT TIME ZONE. Adjust the timestamp value in the RECEIVED column to UTC by subtracting the value of the CURRENT TIME ZONE special register.

```
SELECT RECEIVED - CURRENT TIME ZONE, SOURCE, SUBJECT, NOTE_TEXT
FROM IN_TRAY;
```

ENCRYPTION PASSWORD

The ENCRYPTION PASSWORD special register specifies the encryption password and the password hint (if one exists) that are used by the encryption and decryption built-in functions.

This special register can only be set, by using the SET ENCRYPTION PASSWORD statement, and cannot be referenced directly. The ENCRYPTION PASSWORD special register contains the value of the password that is used by the ENCRYPTION and DECRYPTION built-in functions to encrypt and decrypt data when a password is not explicitly specified as a function argument. The ENCRYPTION PASSWORD special register can also contain a password hint which is associated with the values that are encrypted using the encryption password. The password hint is a character string that is used to help in remembering the password. The GETHINT function is used to return the password hint for an encrypted value.

The initial value of the ENCRYPTION PASSWORD special register is the empty string (' ').

The initial value of the ENCRYPTION PASSWORD special register in a user-defined function or procedure is inherited from the invoking application. In other contexts, the initial value of the special register is the empty string.

The password is not related to DB2 authentication and is used only for data encryption.

Related reference:

"SET ENCRYPTION PASSWORD" on page 1919

SESSION_USER

SESSION_USER specifies the primary authorization ID of the process.

The data type is VARCHAR(128).

If SESSION_USER is referred to in an SQL statement that is executed at a remote DB2 and the primary authorization ID has been translated to a different authorization ID, SESSION_USER specifies the translated authorization ID. For an explanation of authorization ID translation, see *DB2 Administration Guide*. The value of SESSION_USER in a user-defined function or stored procedure is determined according to the rules in Table 40 on page 205.

USER can be specified as a synonym for SESSION_USER.

Example: Display information about tables, views, and aliases that are owned by the primary authorization ID of the process.

```
SELECT * FROM SYSIBM.SYSTABLES WHERE CREATOR = SESSION_USER;
```

SESSION TIME ZONE

The SESSION TIME ZONE special register specifies a value that identifies the time zone of the application process.

The data type is VARCHAR(128).

The time zone value is in the format of ±*th*:*tm*. *th* represents the time zone hour offset. *tm* represents the time zone minute offset. Valid values for *th* are between -12 and +14. Valid values for *tm* are between 0 and 59. SESSION TIMEZONE can be specified as an alternative to SESSION TIME ZONE.

The initial value of the special register in a user-defined function or stored procedure is inherited according to the rules in "Special registers in a user-defined function or a stored procedure" on page 205. In other contexts the initial value of the special register represents the same time zone as the CURRENT TIME ZONE special register.

The value of the special register can be changed by executing the SET SESSION TIME ZONE statement. After a SET SESSION TIME ZONE statement has been processed, the values of the SESSION TIME ZONE and CURRENT TIME ZONE special register might not reflect the same value.

Example: Set the session time zone to '-8:00':

```
SET SESSION TIME ZONE = '-8:00';
```

Related reference:

"SET SESSION TIME ZONE" on page 1927

USER

USER specifies the primary authorization ID of the process. The data type is VARCHAR(128). SESSION_USER is the preferred spelling.

If USER is referred to in an SQL statement that is executed at a remote DB2 and the primary authorization ID has been translated to a different authorization ID, USER specifies the translated authorization ID. For an explanation of authorization ID translation, see *DB2 Administration Guide*. The value of USER in a user-defined function or stored procedure is determined according to the rules in Table 40 on page 205.

Example: Display information about tables, views, and aliases that are owned by the primary authorization ID of the process.

```
SELECT * FROM SYSIBM.SYSTABLES WHERE CREATOR = USER;
```

Special registers in a user-defined function or a stored procedure

You can use all special registers in a user-defined function or a stored procedure. However, you can modify only some of those special registers.

After a user-defined function or a stored procedure completes, DB2 restores all special registers to the values they had before invocation.

The following table shows information that you need when you use special registers in a user-defined function or stored procedure.

Table 40. Characteristics of special registers in a user-defined function or a stored procedure

Special register	Initial value when INHERIT SPECIAL REGISTERS option is specified	Initial value when DEFAULT SPECIAL REGISTERS option is specified	Routine can use SET statement to modify?
CURRENT APPLICATION COMPATIBILITY	The value of bind option APPLCOMPAT for the user-defined function or stored procedure package	The value of bind option APPLCOMPAT for the user-defined function or stored procedure package	Yes
CURRENT APPLICATION ENCODING SCHEME	The value of bind option ENCODING for the user-defined function or stored procedure package	The value of bind option ENCODING for the user-defined function or stored procedure package	Yes
CURRENT CLIENT_ACCTNG	Inherited from the invoking application	Inherited from the invoking application	Not applicable[5]
CURRENT CLIENT_APPLNAME	Inherited from the invoking application	Inherited from the invoking application	Not applicable[5]
CURRENT CLIENT_USERID	Inherited from the invoking application	Inherited from the invoking application	Not applicable[5]
CURRENT CLIENT_WRKSTNNAME	Inherited from the invoking application	Inherited from the invoking application	Not applicable[5]
CURRENT DATE	New value for each SQL statement in the user-defined function or stored procedure package[1]	New value for each SQL statement in the user-defined function or stored procedure package[1]	Not applicable[5]
CURRENT DEBUG MODE	Inherited from the invoking application	DISALLOW	Yes
CURRENT DECFLOAT ROUNDING MODE	Inherited from the invoking application	The value of bind option ROUNDING for the user-defined function or stored procedure package	Yes
CURRENT DEGREE	CURRENT DEGREE[2]	The value of field CURRENT DEGREE on installation panel DSNTIP8	Yes
CURRENT EXPLAIN MODE	Inherited from the invoking application	NO	Yes
CURRENT GET_ACCEL_ARCHIVE	Inherited from the invoking application	System default value	Yes
CURRENT LOCALE LC_CTYPE	Inherited from the invoking application	The value of field CURRENT LC_CTYPE on installation panel DSNTIPF	Yes

Table 40. Characteristics of special registers in a user-defined function or a stored procedure (continued)

Special register	Initial value when INHERIT SPECIAL REGISTERS option is specified	Initial value when DEFAULT SPECIAL REGISTERS option is specified	Routine can use SET statement to modify?
CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION	Inherited from the invoking application	System default value	Yes
CURRENT MEMBER	New value for each SET *host-variable*=CURRENT MEMBER statement	New value for each SET *host-variable*=CURRENT MEMBER statement	Not applicable[5]
CURRENT OPTIMIZATION HINT	The value of bind option OPTHINT for the user-defined function or stored procedure package or inherited from the invoking application[6]	The value of bind option OPTHINT for the user-defined function or stored procedure package	Yes
CURRENT PACKAGE PATH	An empty string if the routine was defined with a COLLID value; otherwise, inherited from the invoking application[4]	An empty string, regardless of whether a COLLID value was specified for the routine[4]	Yes
CURRENT PACKAGESET	Inherited from the invoking application[3]	Inherited from the invoking application[3]	Yes
CURRENT PATH	The value of bind option PATH for the user-defined function or stored procedure package or inherited from the invoking application[6]	The value of bind option PATH for the user-defined function or stored procedure package	Yes
CURRENT PRECISION	Inherited from the invoking application	The value of field DECIMAL ARITHMETIC on installation panel DSNTIP4	Yes
CURRENT QUERY ACCELERATION	Inherited from the invoking application	System default value	Yes
CURRENT REFRESH AGE	Inherited from the invoking application	System default value	Yes
CURRENT ROUTINE VERSION	Inherited from the invoking application	The empty string	Yes
CURRENT RULES	Inherited from the invoking application	The value of bind option SQLRULES for the plan that invokes a user-defined function or stored procedure	Yes
CURRENT SCHEMA	Inherited from the invoking application	The value of CURRENT SCHEMA when the routine is entered	Yes
CURRENT SERVER	Inherited from the invoking application	Inherited from the invoking application	Yes
CURRENT SQLID	The primary authorization ID of the application process or inherited from the invoking application[7]	The primary authorization ID of the application process	Yes[8]
CURRENT TEMPORAL BUSINESS_TIME	Inherited from the invoking application	NULL	Yes
CURRENT TEMPORAL SYSTEM_TIME	Inherited from the invoking application	NULL	Yes

Table 40. Characteristics of special registers in a user-defined function or a stored procedure (continued)

Special register	Initial value when INHERIT SPECIAL REGISTERS option is specified	Initial value when DEFAULT SPECIAL REGISTERS option is specified	Routine can use SET statement to modify?
CURRENT TIME	New value for each SQL statement in the user-defined function or stored procedure package[1]	New value for each SQL statement in the user-defined function or stored procedure package[1]	Not applicable[5]
CURRENT TIMESTAMP	New value for each SQL statement in the user-defined function or stored procedure package[1]	New value for each SQL statement in the user-defined function or stored procedure package[1]	Not applicable[5]
CURRENT TIMESTAMP WITH TIME ZONE	New value for each SQL statement in the user-defined function or stored procedure package[1]	New value for each SQL statement in the user-defined function or stored procedure package[1]	Not applicable[5]
CURRENT TIME ZONE	Inherited from the invoking application	Inherited from the invoking application	Not applicable[5]
ENCRYPTION PASSWORD	Inherited from the invoking application	Inherited from the invoking application	Yes
SESSION TIME ZONE	Inherited from the invoking application	The value of CURRENT TIME ZONE when the routine is entered	Yes
SESSION_USER or USER	Primary authorization ID of the application process	Primary authorization ID of the application process	Not applicable[5]

Notes:

1. If the user-defined function or stored procedure is invoked within the scope of a trigger, DB2 uses the timestamp for the triggering SQL statement as the timestamp for all SQL statements in the package.
2. DB2 allows parallelism at only one level of a nested SQL statement. If you set the value of the CURRENT DEGREE special register to ANY, and parallelism is disabled, DB2 ignores the CURRENT DEGREE value.
3. If the routine definition includes a specification for COLLID, DB2 sets CURRENT PACKAGESET to the value of COLLID. If both CURRENT PACKAGE PATH and COLLID are specified, the CURRENT PACKAGE PATH value takes precedence and COLLID is ignored.
4. If the function definition includes a specification for PACKAGE PATH, DB2 sets CURRENT PACKAGE PATH to the value of PACKAGE PATH.
5. Not applicable because no SET statement exists for the special register.
6. If a program within the scope of the invoking program issues a SET statement for the special register before the user-defined function or stored procedure is invoked, the special register inherits the value from the SET statement. Otherwise, the special register contains the value that is set by the bind option for the user-defined function or stored procedure package.
7. If a program within the scope of the invoking program issues a SET CURRENT SQLID statement before the user-defined function or stored procedure is invoked, the special register inherits the value from the SET statement. Otherwise, CURRENT SQLID contains the authorization ID of the application process.
8. If the user-defined function or stored procedure package uses a value other than RUN for the DYNAMICRULES bind option, the SET CURRENT SQLID statement can be executed. However, it does not affect the authorization ID that is used for the dynamic SQL statements in the package. The DYNAMICRULES value determines the authorization ID that is used for dynamic SQL statements.

Related concepts:

DYNAMICRULES bind option (DB2 Application programming and SQL)

Related reference:

BIND and REBIND options (DB2 Commands)

"Special registers" on page 156

Column names

The meaning of a column name depends on its context.

A column name can be used to:

- Declare the name of a column, as in a CREATE TABLE statement.
- Specify the name of a column, as in a CREATE FUNCTION statement to name a column of the result table of a table function.
- Identify a column, as in a CREATE INDEX statement.
- Specify values of the column, as in the following contexts:
  - In an *aggregate function*, a column name specifies all values of the column in the group or intermediate result table to which the function is applied. (Groups and intermediate result tables are explained in Chapter 4, "Queries," on page 761.) For example, MAX(SALARY) applies the function MAX to all values of the column SALARY in a group.
  - In a *GROUP BY* or *ORDER BY* clause, a column name specifies all values in the intermediate result table to which the clause is applied. For example, ORDER BY DEPT orders an intermediate result table by the values of the column DEPT.
  - In an *expression*, a *search condition*, or a *scalar function*, a column name specifies a value for each row or group to which the construct is applied. For example, when the search condition CODE = 20 is applied to some row, the value specified by the column name CODE is the value of the column CODE in that row.
- Provide a column name for an expression to temporarily rename a column, as in the *correlation-clause* of a *table-reference* in a FROM clause or as in the AS clause in the *select-clause*.

Qualified column names

A qualifier for a column name can be a table name, a view name, an alias name, a synonym, or a correlation name. Whether a column name can be qualified depends, like its meaning, on its context.

- In some forms of the COMMENT and LABEL statements, a column name must be qualified. This is shown in the syntax diagrams.
- Where the column name specifies values of the column, a column name can be qualified at the user's option.
- In the column list of an INSERT statement, a column name can be qualified.
- In the *assignment-clause* of an UPDATE or a MERGE statement, a column name can be qualified.
- In all other contexts, a column name must not be qualified. This rule will be mentioned in the discussion of each statement to which it applies.

Where a qualifier is optional, it can serve two purposes. See "Column name qualifiers to avoid ambiguity" and "Column name qualifiers in correlated references" on page 211 for details.

Correlation names

A *correlation name* can be defined in the FROM clause of a query and after the name of the target table or view in an UPDATE, MERGE, or DELETE statement.

For example, the following clause establishes Z as a correlation name for X.MYTABLE:

```
FROM X.MYTABLE Z
```

With Z defined as a correlation name for table X.MYTABLE, only Z should be used to qualify a reference to a column of X.MYTABLE in that SELECT statement.

A correlation name is associated with a table, view, nested table expression or table function only within the context in which it is defined. Hence, the same correlation name can be defined for different purposes in different statements. In a nested table expression or table function, a correlation name is required.

As a qualifier, a correlation name can be used to avoid ambiguity or to establish a correlated reference. It can also be used merely as a shorter name for a table or view. In the example, Z might have been used merely to avoid having to enter X.MYTABLE more than once.

Names that are specified in a FROM clause are either *exposed* or *non-exposed*. A correlation name is always an exposed name. A table name or view name is said to be exposed in that FROM clause if a correlation name is not specified. For example, in the following FROM clause, a correlation name is specified for EMPLOYEE, but not for DEPARTMENT; therefore, DEPARTMENT is an exposed name, and EMPLOYEE is not an exposed name:

```
FROM EMPLOYEE E, DEPARTMENT
```

The use of a correlation name in the FROM clause also allows the option of specifying a list of column names to be associated with the columns of the result table. As with a correlation name, the listed column names should be the names that are used to reference the columns in that SELECT statement. For example, assume that the name of the first column in the DEPT table is DEPTNO. Given this FROM clause in a SELECT statement:

```
  FROM DEPT D (NUM,NAME,MGR,ANUM,LOC)
```

You should use D.NUM instead of D.DEPTNO to reference the first column of the table.

If a list of columns is specified, it must consist of as many names as there are columns in the *table-reference*. Each column must be unique and unqualified.

Column name qualifiers to avoid ambiguity

In the context of a function, a GROUP BY clause, an ORDER BY clause, an expression, or a search condition, a column name refers to values of a column in some table or view in a DELETE or UPDATE statement or *table-reference* in a FROM clause.

The tables, views, and *table-references*[12] that might contain the column are called the *object tables* of the context. Two or more object tables might contain columns with the same name. One reason for qualifying a column name is to designate the object from which the column comes. For information on avoiding ambiguity between SQL parameters and variables and column names, see "References to SQL parameters and SQL variables" on page 2033.

A nested table expression which is preceded by a TABLE keyword will consider *table-references* that precede it in the FROM clause as object tables. The *table-references* that follow it are not considered as object tables.

Table designators: A qualifier that designates a specific object table is called a *table designator*. The clause that identifies the object tables also establishes the table designators for them. For example, the object tables of an expression in a SELECT statement are named in the FROM clause that follows it, as in the following statement:

```
SELECT DISTINCT Z.EMPNO, EMPTIME, PHONENO
  FROM DSN8B10.EMP Z, DSN8B10.EMPPROJACT
  WHERE WORKDEPT = 'D11'
    AND EMPTIME > 0.5
    AND Z.EMPNO = DSN8B10.EMPPROJACT.EMPNO;
```

Table designators in the FROM clause are established as follows:

- A name that follows a table or view name is both a correlation name and a table designator. Thus, Z is a table designator and qualifies the first column name in the select list.
- An exposed table or view name is a table designator. Thus, the qualified table name, Thus, the qualified table name, DSN8B10.EMPPROJACT is a table designator and qualifies the second column name in the select list.

Two or more object tables can be instances of the same table. In this case, distinct correlation names must be used to unambiguously designate the particular instance of the table. In the following example, the X and Y in the FROM clause are defined to refer, respectively, to the first and second instances of the DSN8B10.EMP table:

```
SELECT *
  FROM DSN8B10.EMP X, DSN8B10.EMP Y;
```

Avoiding undefined or ambiguous references in DB2 SQL: When a column name refers to values of a column, the following situations result in errors:

- No object table contains a column with the specified name. The reference is undefined.
- The column name is qualified by a table designator, but the table named does not include a column with the specified name. Again, the reference is undefined.
- The name is unqualified and more than one object table includes a column with that name. The reference is ambiguous.

Avoid ambiguous references by qualifying a column name with a uniquely defined table designator. If the column is contained in several object tables with different names, the table names can be used as designators. Ambiguous references can also be avoided without the use of the table designator by giving unique names to the columns of one of the object tables using the column name list following the correlation name.

12. In the case of a *joined-table*, each *table-reference* within the *joined-table* is an object table.

Two or more object tables can be instances of the same table. A FROM clause that includes *n* references to the same table should include at least *n* - 1 unique correlation names.

For example, in the following FROM clause X and Y are defined to refer, respectively, to the first and second instances of the table EMP.

```
SELECT X.LASTNAME, Y.LASTNAME
  FROM DSN8B10.EMP X, DSN8B10.EMP Y
  WHERE Y.JOB = 'MANAGER'
    AND X.WORKDEPT = Y.WORKDEPT
    AND X.JOB <> 'MANAGER';
```

When qualifying a column with the exposed table name form of a table designator, either the qualified or unqualified form of the exposed table name can be used. However, the qualifier used and the table used must be the same after fully qualifying the table name or view name and the table designator.

Example 1: If the authorization ID of the statement is CORPDATA, the following statement is valid:

```
SELECT CORPDATA.EMPLOYEE.WORKDEPT
  FROM EMPLOYEE;
```

Example 2: If the authorization ID of the statement is REGION, the following statement is invalid because EMPLOYEE represents the table REGION.EMPLOYEE, but the qualifier for WORKDEPT represents a different table, CORPDATA.EMPLOYEE:

```
SELECT CORPDATA.EMPLOYEE.WORKDEPT              -- Incorrect
  FROM EMPLOYEE;
```

Example 3: If the authorization ID of the statement is REGION, the following statement is invalid because EMPLOYEE in the select list represents the table REGION.EMPLOYEE, but the explicitly qualified table name in the FROM clause represents a different table, CORPDATA.EMPLOYEE.

```
SELECT EMPLOYEE.WORKDEPT                      -- Incorrect
  FROM CORPDATA.EMPLOYEE;
```

Column name qualifiers in correlated references

A reference to a column of a table identified at a higher level is called a *correlated reference*. Because the same table or view can be identified at many levels, unique correlation names are recommended as table designators. It is good practice to use these unique correlation names to qualify column names.

A *subselect* is a form of a query that can be used as a component of various SQL statements. A *subquery* is a form of a fullselect that is enclosed within parenthesis. For example, a subquery can be used in a search condition. A fullselect that is used to retrieve a single value as an expression within a statement is called a *scalar fullselect* or a *scalar subquery*. A fullselect that is used in the FROM clause of a query is called a *nested table expression*.

A subquery can include search conditions of its own, and these search conditions can, in turn, include subqueries. Thus, an SQL statement can contain a hierarchy of subqueries. Those elements of the hierarchy that contain subqueries are said to be at a higher level than the subqueries they contain.

Every element of the hierarchy has a clause that establishes one or more table designators. This is the FROM clause, except in the highest level of an UPDATE,

where it is the table or view being updated. A search condition of a subquery can reference not only columns of the tables identified by the FROM clause of its own element of the hierarchy, but also columns of tables identified at any level along the path from its own element to the highest level of the hierarchy.

A correlated reference to column C of table T can be of the form C, T.C, or Q.C, if Q is a correlation name defined for T. However, **a correlated reference in the form of an unqualified column name is not good practice**. The following explanation is based on the assumption that a correlated reference is always in the form of a qualified column name and that the qualifier is a correlation name.

A qualified column name, Q.C, is a correlated reference only if these three conditions are met:
- Q.C is used in a search condition or in a select list of a subquery.
- Q does not name a table used in the FROM clause of that subquery.
- Q does name a table used at some higher level.

Q.C refers to column C of the table or view at the level where Q is used as the table designator of that table or view. Because the same table or view can be identified at many levels, unique correlation names are recommended as table designators. If Q is used to name a table at more than one level, Q.C refers to the lowest level that contains the subquery that includes Q.C.

If a correlation name is defined as the table designator of the table or view, but the table or view name is used as the column qualifier instead of the correlation name, an error is returned.

For example, in the following statement, the correlated reference X.WORKDEPT (in the last line) refers to the value of WORKDEPT in table DSN8B10.EMP at the level of the first FROM clause (which establishes X as a correlation name for DSN8B10.EMP.). The statement lists employees who make less than the average salary for their department.

```
SELECT EMPNO, LASTNAME, WORKDEPT
  FROM DSN8B10.EMP X
  WHERE SALARY < (SELECT AVG(SALARY)
                    FROM DSN8B10.EMP
                    WHERE WORKDEPT = X.WORKDEPT);
```

The following example shows a correlated reference in the select list of the subquery.

```
SELECT T1.KEY1
  FROM BP1TBL T1
  GROUP BY T1.KEY1
  HAVING MAX(T1.KEY1) = (SELECT MIN(T1.KEY1) + MIN(T2.KEY1)
                    FROM BP2TBL T2);
```

Related concepts:

Chapter 4, “Queries,” on page 761

Resolution of column name qualifiers and column names

The rules for resolving column name qualifiers apply to every SQL statement that includes a subselect and are applied before synonyms and aliases are resolved.

Names in a FROM clause are either *exposed* or *non-exposed*. A correlation name for a table name, view name, nested table expression, or reference to a table function is always exposed. A table name or a view name that is not followed by a correlation name is also exposed.

Although DB2 for z/OS does not enforce this rule strictly, in IBM SQL and ANSI/ISO SQL, the exposed names in a FROM clause must be unique, and the qualifier of a column name must be an exposed name. Therefore, for good programming practices, ensure that all exposed names are unique and that all qualified column names are qualified with the appropriate exposed name.

The rules for finding the referent of a column name qualifier are as follows:

1. Let Q be a one-, two-, or three-part name, and let Q.C denote a column name in subselect S. Q must designate a table or view identified in the statement that includes S and that table or view must have a column named C. An additional requirement differs for two cases:
   - If Q.C *is not* in a *search-condition* or S *is not* a subquery, Q must designate a table or view identified in the FROM clause of S. For example, if Q.C is in a SELECT clause, Q refers to a table or view in the following FROM clause.
   - If Q.C *is* in a *search-condition* and S *is* a subquery, Q must designate a table or view identified either in the FROM clause of S or in a FROM clause of a subselect that directly or indirectly includes S. For example, if Q.C is in a WHERE clause and S is the only subquery in the statement, the table or view that Q refers to is either in the FROM clause of S or the FROM clause of the subselect that includes S.
2. The same table or view can be identified more than once in the same statement. The particular occurrence of the table or view that Q refers to is determined by a procedure equivalent to the following steps:
   a. The one- and two-part names in every FROM clause and the one- and two-part qualifiers of column names are expanded into a fully-qualified form.

      For example, if a dynamic SQL statement uses FROM Q and DYNAMICRULES run behavior (RUN) is in effect, Q is expanded to S.A.Q, where S is the value of CURRENT SERVER and A is the value of CURRENT SCHEMA. (If DYNAMICRULES bind behavior is in effect instead, A is the plan or package qualifier as determined during the bind process or the qualifier for the native SQL procedure as determined when the procedure was defined.) This step is later referred to as "name completion". An error occurs if the first part of every name (the location) is not the same.
   b. Q, now a three-part name, is compared with every name in the FROM clause of S. If Q.C is in a *search-condition* and S is a subquery, Q is next compared with every name in the FROM clause of the subselect that contains S. If that subselect is a subquery, Q is then compared with every name in the FROM clause of the subselect containing that subquery, and so on. If a FROM clause includes multiple names, the comparisons in that clause are made in order from left to right.
   c. The referent of Q is selected by these rules:
      - If Q matches exactly one name, that name is selected.
      - If Q matches more than one name, but only one exposed name, that exposed name is selected.
      - If Q matches more than one exposed name, the first of those names is selected.
      - If Q matches more than one name, none of which are exposed names, the first of those names is selected.

      If Q does not match any name, or if the table or view designated by Q does not include a column named C, an error occurs.

d. Otherwise, Q.C is resolved to column C of the occurrence of the table or view identified by the selected name.

3. A warning occurs for any of these cases:
   - The selected name is not an exposed name.
   - The selected name is an exposed name that has an unexposed duplicate that appears before the selected name in the ordered list of names to which Q is compared.
   - The selected name is an exposed name that has an exposed duplicate in the same FROM clause.
   - Another name would have been selected had the matching been performed before name completion.

   The rules for resolving column name qualifiers apply to every SQL statement that includes a subselect and are applied before synonyms and aliases are resolved. In the case of a searched UPDATE or DELETE statement, the first clause of the statement identifies the table or view to be updated or deleted. That clause can include a correlation name and, with regard to name resolution, is equivalent to the first FROM clause of a SELECT statement. For example, a subquery in the search condition of an UPDATE statement can include a correlated reference to a column of the updated rows.

   The rules for column names in the ORDER BY clause are the same as other clauses.

References to variables

A *variable* in an SQL statement specifies a value that can be changed when the SQL statement is executed. There are several types of variables used in SQL statements.

host variable
Host variables are defined by statements of a host language. For more information about how to refer to host variables, see "References to host variables" on page 215.

transition variable
Transition variables are defined in a trigger and refer to either the old or new values of columns of the subject table of a trigger. For more information about how to refer to transition variables, see "CREATE TRIGGER" on page 1482.

SQL variable
SQL variables are defined by an SQL compound statement in an SQL function or SQL procedure. For more information about SQL variables, see "References to SQL parameters and SQL variables" on page 1964.

SQL parameter
SQL parameters are defined in an CREATE FUNCTION (SQL Scalar) or CREATE PROCEDURE (SQL) statement. For more information about SQL parameters, see "References to SQL parameters and SQL variables" on page 1964.

parameter marker
Parameter markers are specified in an SQL statement that is dynamically prepared instead of host variables. For more information about parameter markers, see Parameter markers. in the PREPARE statement.

Unless otherwise noted, the term *host variable* in syntax diagrams is used to describe where a host variable, transition variable, SQL variable, SQL parameter, or parameter marker can be used.

References to host variables

Host variables are defined directly by statements of the host language or indirectly by SQL extensions. A *host-variable* in an SQL statement must identify a host variable that is described in the program according to the rules for declaring host variables. Host variables cannot be referenced in dynamic SQL statements; parameter markers must be used instead.

A *host variable* is either of these items that is referred to in an SQL statement:

- A variable in a host language such as a PL/I variable, C variable, Fortran variable, REXX variable, Java variable, COBOL data item, or Assembler language storage area
- A host language construct that was generated by an SQL precompiler from a variable declared using SQL extensions

Host variables are defined directly by statements of the host language or indirectly by SQL extensions as described in *DB2 Application Programming and SQL Guide*. Host variables cannot be referenced in dynamic SQL statements; parameter markers must be used instead. For more information about parameter markers, see "Host variables in dynamic SQL" on page 217.

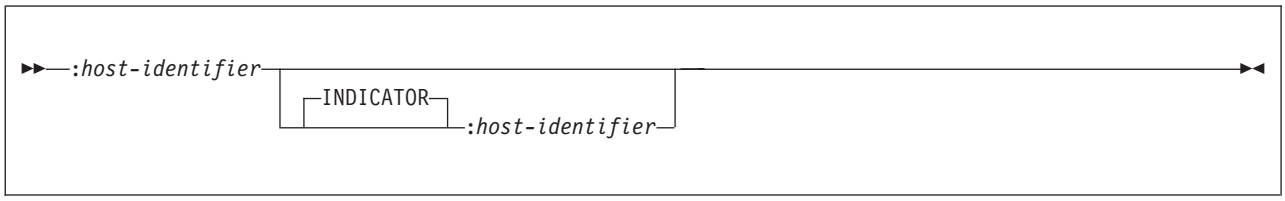
A host-variable in an SQL statement must identify a host variable that is described in the program according to the rules for declaring host variables.

In PL/I, C, and COBOL, host variables can be referred to in ways that do not apply to Fortran and Assembler language. This is explained in "Host structures in PL/I, C, and COBOL" on page 229. The following applies to all host languages.

The term *host-variable*, as used in the syntax diagrams, shows a reference to a host variable. In a SET *host variable* statement and the INTO clause of a FETCH, SELECT INTO, or VALUES INTO statement, a host variable is an output variable to which a value is assigned by DB2. In a CALL statement, a host variable can be an output argument that is assigned a value after execution of the procedure, an input argument that provides an input value for the procedure, or both an input and output argument. In all other contexts, a host variable is an input variable which provides a value to DB2.

Non-Java variable references

The general form of a host variable reference in all languages other than Java is:



Each host identifier must be declared in the source program, except in a program written in REXX. The first host identifier designates the main variable; the second host identifier designates its indicator variable. The variable designated by the second host identifier must be a small integer. Indicator variables appear in two forms, normal indicator variables and extended indicator variables.

The purposes of normal indicator variable are to:

- Specify a non-null value. A 0 (zero), or positive value of the indicator variable specifies that the associated, first *host-identifier* provides the value of this host variable reference.
- Specify the null value. A negative value of the indicator variable specifies the null value.

In addition, on output, an indicator variable can indicate the following :

- A numeric conversion error (such as a divide by 0 or overflow) has occurred. A value of -2 for the indicator variable indicates a null result because of either numeric truncation or arithmetic warnings.
- A character could not be converted. A value of -2 for the indicator variable indicates a null result because of character string conversion warnings.
- No value was returned. A value of -3 for the indicator variable indicates a null result because the current row of the cursor is on a hole that is detected during a multiple row FETCH.
- Report the original length of a truncated string, if the string is not a LOB.
- Report the seconds portion of a time if the time is truncated on assignment to a host variable.

Extended indicator variables are limited to the input of host variables, and can specify the following:

- A non-null value. A 0 (zero), or positive value specifies that the associated, first *host-identifier* provides the value of this host variable reference.
- The null value. A -1, -2, -3, -4, or -6 value specifies the null value.
- The default value. A -5 value specifies that the target column for this host variable is to be set to its default value.
- An unassigned value. A -7 value specifies that the target column for this host variable is to be treated as if it had not been specified in the statement.

Extended indicator variables are only enabled if requested, and all indicator variables are otherwise normal indicator variables. Extended indicator variables are enabled when EXTENDEDINDICATOR(YES) is used, or when the **WITH EXTENDED INDICATORS** prepare attribute has been specified for the statement. In comparison to normal indicator variables, extended indicator variables have no additional restrictions for where the values for null and non-null can be used. There are no restrictions against using extended indicator variable values in indicator structures with host structures. There are no restrictions that result from the use of extended indicator variable values with host arrays in multiple-row statements. Restrictions on where the extended indicator variable values of default and unassigned are allowed apply uniformly, no matter how they are represented in the host application. The default and unassigned extended indicator variable values can only appear in limited, specified uses. Output indicator variables are never extended indicator variables.

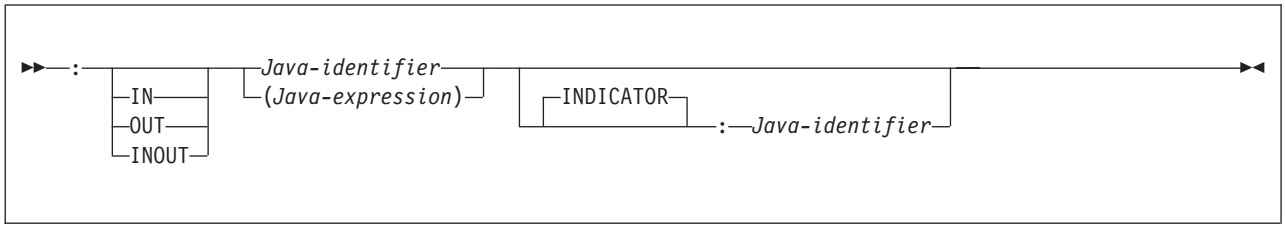
When extended indicator variables are enabled, there are no restrictions against use of 0 (zero), or positive indicator variable values. However, negative indicator variable values outside the range -1 through -7 must not

be specified. When extended indicator variables are enabled, the default and unassigned extended indicator values must not appear in contexts in which they are not supported.

When extended indicator variables are enabled, if the value of an extended indicator variable is greater than or equal to zero, the data type of the input host variable must be compatible with the data type of the target column. If the value of an extended indicator variable is less than zero, DB2 does not test for data type compatibility between the input host variable and the target column.

Java variable references

The general form of a host variable reference in Java is:



Each Java-identifier must be declared in the source program. The variable designated by the second Java-identifier is called an indicator variable and must be a short.

In Java, indicator variables are not always needed. Instead, instances of a Java class can be set to a null value. Variables defined as Java primitive types can not be set to a null value. When using an extended indicator variable, or when using a Java primitive type in assigning a null value or where the Java primitive type might be assigned null on output, indicator variables must be used.

If IN, OUT, or INOUT is not specified, the default depends on the context in which the variable is used. If the Java variable is used in an INTO clause, OUT is the default. Otherwise, IN is the default.

An SQL statement that refers to host variables must be within the scope of the declaration of those host variables. For host variables referred to in the SELECT statement of a cursor, the OPEN statement, and the DECLARE CURSOR statement have to be in the same scope.

All references to host variables must be preceded by a colon. If an SQL statement references a host variable without a preceding colon, the precompiler issues an error for the missing colon or interprets the host variable as an unqualified column name, which might lead to unintended results. The interpretation of a host variable without a colon as a column name occurs when the host variable is referenced in a context in which a column name can also be referenced.

Host variables in dynamic SQL

In dynamic SQL statements, parameter markers are used instead of host variables.

A *parameter marker* is a question mark (?) that represents a position in a dynamic SQL statement where the application will provide a value; that is, where a host variable would be found if the statement string were a static SQL statement. The following examples show a static SQL statement that uses host variables and a dynamic statement that uses parameter markers:

```
INSERT INTO DEPT VALUES (:HV_DEPTNO, :HV_DEPTNAME, :HV_MGRNO, :HV_ADMRDEPT)
INSERT INTO DEPT VALUES (?, ?, ?, ?)
```

For more information on parameter markers, see Parameter markers under the PREPARE statement.

References to LOB host variables

Regular LOB variables (CLOB, DBCLOB, and BLOB), LOB locator variables and LOB file reference variables can be defined in all host languages with a few exceptions.

Where LOBs are allowed, the term *host-variable* in a syntax diagram can refer to a regular host variable, a locator variable, or a file reference variable. Since these variables are not native data types in host programming languages, SQL extensions are used and the precompilers generate the host language constructs necessary to represent each variable.

- REXX supports LOB locators and LOB file reference variables.
- Java supports LOBs and LOB file references, but not LOB locators.

When it is possible to define a host variable that is large enough to hold an entire LOB value and the performance benefit of delaying the transfer of data from the server is not required, a LOB locator or LOB file reference is not needed. However, it is often not acceptable to store an entire LOB value in temporary storage due to host language restrictions, storage restrictions, or performance requirements. When storing an entire LOB value at one time is not acceptable, a LOB value can be referenced using a LOB locator and portions of the LOB value can be accessed or the entire LOB value can be stored in a file and a LOB file reference can be used to access the LOB data.

References to LOB locator variables

A LOB locator variable is a host variable that contains the locator representing a LOB value on the database server.

A locator variable in an SQL statement must identify a LOB locator variable described in the program according to the rules for declaring locator variables. This is always indirectly through an SQL statement. For example, in C:

```
static volatile SQL TYPE IS CLOB_LOCATOR *loc1;
```

The term *locator-variable*, as used in the syntax diagrams, shows a reference to a LOB locator variable. The meta-variable *locator-variable* can be expanded to include a *host-identifier* the same as that for *host-variable*.

Like all other host variables, a LOB locator variable can have an associated indicator variable. Indicator variables for LOB locator variables behave in the same way as indicator variables for other data types. When a null value is returned from the database, the indicator variable is set and the locator host variable is unchanged. This means a locator can never represent a null value. However, when the indicator variable associated with a LOB locator is null, the value of the referenced LOB value is null.

If a locator variable does not currently represent any value, an error occurs when the locator variable is referenced.

When a transaction commits, LOB locators that were acquired by the transaction are released unless a HOLD LOCATOR statement was issued for the LOB locator. When the transaction ends, all LOB locators are released.

It is the application programmer's responsibility to guarantee that any LOB locator is used only in SQL statements that are executed at the same server that originally generated the LOB locator. For example, assume that a LOB locator is returned from one server and assigned to a LOB locator variable. If that LOB locator variable is subsequently used in an SQL statement that is executed at a different server unpredictable results will occur.

References to XML host variables

XML host variables can be declared as another host variable type. This allows XML data to be declared in host languages that do not support XML data as a native data type.

XML host variables can be declared as the following host variable types:

- XML AS CLOB(*n*)

  Declares a CLOB host variable that contains XML data that is encoded in the CCSID for the host variable.
- XML AS DBCLOB(*n*)

  Declares a DBCLOB host variable that contains XML data that is encoded in the graphic CCSID for the host variable.
- XML AS BLOB(*n*)

  Declares a BLOB host variable that contains XML data that is encoded as specified within the data according to the XML 1.0 specification for determining encoding.
- XML AS CLOB_FILE

  Declares a CLOB file reference variable that contains XML data that is encoded in the CCSID for the file reference variable.
- XML AS DBCLOB_FILE

  Declares a DBCLOB file reference variable that contains XML data that is encoded in the CCSID for the file reference variable.
- XML AS BLOB_FILE

  Declares a BLOB file reference variable that contains XML data that is encoded in the CCSID for the file reference variable.

See “References to file reference variables” on page 220 for additional information about file reference variables.

Although the application XML host variable declaration includes a LOB type specification, the host variable declarations all map to the XML data type, not the LOB type that is used in the application declaration. The application might also use non-XML host variables in place of XML host variables. For example, when a prepared statement is executed, the application might use a character host variable to replace an XML parameter marker in the statement.

Although the XML data type is incompatible with all other data types, both XML and non-XML data types can be used for input to and output from XML data. Applications can use either XML host variables, character host variables, or binary string host variables to input to and output from XML data by using SQL statements.

The following table summarizes the conversions built-in data types (including XML) to and from the supported host variable data types within embedded applications. The built-in data types are specified in the rows. A Y indicates that the built-in data type can be assigned to or assigned from the host variable type.

Table 41. Application host variable compatibility with the built-in data types for applications that contain embedded SQL

	application host variable data type					
built-in data type	**CHAR, VARCHAR, CLOB, CLOB_FILE**	**GRAPHIC, VARGRAPHIC, DBCLOB, DBCLOB_FILE**	**BINARY, VARBINARY, BLOB, BLOB_FILE**	**XML AS CLOB, XML AS CLOB_FILE**	**XML AS DBCLOB, XML AS DBCLOB_FILE**	**XML AS BLOB, XML AS BLOB_FILE**
CHAR	Y	Y				
VARCHAR	Y	Y				
CLOB	Y	Y				
GRAPHIC	Y	Y				
VARGRAPHIC	Y	Y				
DBCLOB	Y	Y				
BINARY			Y			
VARBINARY			Y			
BLOB			Y			
XML	Y	Y	Y	Y	Y	Y

References to file reference variables

File reference variables and arrays are used for direct file input and output for LOB and XML values (when the XML value is declared using XML AS *variable-type*), and can be defined in all host languages.

Since these are not native data types, SQL extensions are used and the DB2 precompiler or coprocessor generates the host language constructs necessary to represent each variable or array. In the case of REXX, LOB values are mapped to strings. See "References to XML host variables" on page 219 for more information about XML host variables.

A file reference variable represents (rather than contains) the file, just as a LOB locator represents, rather than contains, the LOB data. Database queries, updates, and inserts can use file reference variables to store or to retrieve single column values. As with all other host variables, file reference variables can have an associated indicator variable.

A file reference variable has the following properties:

Data type
BLOB, CLOB, or DBCLOB. This property is specified when the variable is declared using BLOB_FILE, CLOB_FILE, or DBCLOB_FILE.

Direction
This must be specified by the application program at run time (it is implicitly specified as part of the File options value). The direction can be either of the following:

- Input

  Input is used as a source of data on an EXECUTE statement, an OPEN statement, an update operation, an insert operation, or a delete operation.
- Output

Output is used as the target of data. For example, on a FETCH statement or a SELECT INTO statement.

File name
This must be specified by the application program at run time. It must be the complete path name of the file. Within an application, a file should only be referenced one time in a file reference variable.

File name length
This must be specified by the application program at run time. It is the length of the file name in bytes.

Data length
Sets the data length to the length of the new data that is written to the file. The length is in bytes. Data length is unused on input.

File options
Options are set by an INTEGER value in a field in the file reference variable structure. One of the following values must be specified in an application for each file reference variable before that file reference variable can be used in the application:

SQL_FILE_READ
This is a regular file that can be opened, read and closed. (The option is SQL-FILE-READ in COBOL, sql_file_read in FORTRAN, and READ in REXX.) SQL_FILE_READ is an input (from client to server) file option.

SQL_FILE_CREATE
Create a new file. If the file already exists, an error is returned. (The option is SQL-FILE-CREATE in COBOL, sql_file_create in FORTRAN, and CREATE in REXX.) SQL_FILE_CREATE is an output (from server to client) file option.

SQL_FILE_OVERWRITE
If an existing file with the specified name exists, it is overwritten; otherwise a new file is created. (The option is SQL-FILE-OVERWRITE in COBOL, sql_file_overwrite in FORTRAN, and OVERWRITE in REXX.). SQL_FILE_OVERWRITE is an output (from server to client) file option.

SQL_FILE_APPEND
If an existing file with the specified name exists, the output is appended to it; otherwise a new file is created. (The option is SQL-FILE-APPEND in COBOL, sql_file_append in FORTRAN, and APPEND in REXX.) SQL_FILE_APPEND is an output (from server to client) file option.

The encoding scheme CCSID of the file name is based on the encoding scheme of the application. The CCSID of the LOB or XML data (the contents of the file) can be set by the application by using a DECLARE *host-variable* CCSID statement if the CCSID of the LOB or XML data is different from the CCSID of the application. DB2 will perform any character conversion that is required prior to the LOB or XML data being inserted into a table or written to a file.

References to stored procedure result sets

An application, written in a programming language other than Java, can access a result set that is returned from a stored procedure. A result set locator is used by the invoking application to access the result set.

A result set locator value for a result set can be obtained from an ASSOCIATE LOCATOR statement or with the DESCRIBE PROCEDURE statement. For more information, see "ASSOCIATE LOCATORS" on page 1111 and "DESCRIBE PROCEDURE" on page 1603.

The result set locator value is specified on an ALLOCATE CURSOR statement to define a cursor in the invoking application and to associate it with a stored procedure result set. For more information, see "ALLOCATE CURSOR" on page 847.

A DESCRIBE CURSOR statement can be used in the invoking application to obtain information on the characteristics of the columns of a stored procedure result set. For more information, see "DESCRIBE CURSOR" on page 1591.

The application can then access the rows of the result set using FETCH statements with the allocated cursor.

References to result set locator variables

A *result set locator variable* is a variable that contains the locator that identifies a stored procedure result set. A result set locator variable in an SQL statement must identify a result set locator variable described in the program according to the rules for declaring result set locator variables. This is always indirectly through an SQL statement.

For example, in C:

```
static volatile SQL TYPE IS RESULT_SET_LOCATOR *loc1;
```

A result set locator variable in an SQL procedure is defined with the RESULT_SET_LOCATOR VARYING in a compound statement. See "compound-statement" on page 1977 for additional information.

The term *rs-locator-variable*, as used in the syntax diagrams, shows a reference to a result set locator variable. The meta-variable *rs-locator-variable* can be expanded to include a *host-identifier* the same as that for *host-variable*.

When the indicator variable associated with a result set locator is null, the referenced result set is not defined.

If a result set locator variable does not currently represent any stored procedure result set, an error occurs when the locator variable is referenced.

A commit operation destroys all open cursors that were declared in the stored procedure without the WITH HOLD option and the result set locators that are associated with those cursors. Otherwise, a cursor and its associated result set locator persist past the commit.

References to built-in global variables

DB2 provides several built-in global variables.

SYSIBMADM.GET_ARCHIVE

Contains a string value that indicates whether a reference to an archive-enabled table in a table-reference should include rows in the associated archive table.

This global variable is defined with the following characteristics:

- It is updatable, with values maintained by the user.
- The type is CHAR(1).
- The schema is SYSIBMADM.
- The scope of this global variable is session.

The global variable can be set to the following:

Y Specifies that when a table-reference is an archive-enabled table, the table reference includes rows in the associated archive table.

If the SYSIBMADM.GET_ARCHIVE global variable is set to 'Y' and the ARCHIVESENSITIVE bind option is set to 'Y', an archive-enabled table cannot be referenced in an inline SQL table function or in the definition of a row permission or column mask that is activated by a data change statement or query.

N Specifies that when a table-reference is an archive-enabled table, the table reference does not include rows in the associated archive table. This is the default value.

Related information:

"table-reference" on page 773

Archive-enabled tables and archive tables (Introduction to DB2 for z/OS)

SYSIBMADM.MOVE_TO_ARCHIVE

Contains a string value that indicates whether the deletion of a row of an archive-enabled table should result in storing a copy of the deleted row in the associated archive table.

This global variable has the following characteristics:

- It is updatable, with values maintained by the user.
- The type is CHAR(1).
- The schema is SYSIBMADM.
- The scope of this global variable is session.

The global variable can be set to the following:

Y Specifies that a delete of a row in an archive-enabled table will result in storing a copy of the deleted row in the associated archive table. Additionally, when the global variable is set to 'Y', an insert or update operation that specifies the archive-enabled table as the target of the statement will return an error.

E Specifies that a delete of a row in an archive-enabled table will result in storing a copy of the deleted row in the associated archive table.

N Specifies that a delete of a row in an archive-enabled table will not result in storing a copy of a deleted row in the associated archive table. This is the default value.

Related information:

Archive-enabled tables and archive tables (Introduction to DB2 for z/OS)

SYSIBM.CLIENT_IPADDR

Contains the value of the client IP address for the connection. For remote client connections, the value is the host IP address of the application that is used to establish the connection. For local host applications, the value is NULL. For remote host applications, the value is the IP address that is associated with the DB2 subsystem used to establish the connection as shown by issuing the -DISPLAY DDF command.

This global variable has the following characteristics:

- The data type is CHAR(39).
- The value is set by DB2 as obtained from the network.
- The value is NULL if the client did not connect with TCP/IP or SSL protocol.
- The format of the client IP address is TCP/IP IPv6 which is left-aligned with a colon separated hexadecimal address:

  ```
  1111:2222:3333:4444:5555:6666:7777:8888
  ```

  Or IPv4 mapped IPv6 to format:

  ```
  ::FFF:9.30.115.135
  ```

Related information:

-DISPLAY DDF (DB2) (DB2 Commands)

DSNL085I (DB2 Messages)

DSNL089I (DB2 Messages)

References to built-in session variables

DB2 provides several built-in session variables that contain information about the server and application process. The value of a built-in session variable can be obtained by invoking the GETVARIABLE function with the name of the built-in session variable.

DB2 provides the following built-in session variables:

SYSIBM.APPLICATION_ENCODING_SCHEME
Contains a string that corresponds to the value that is specified for the APPLICATION ENCODING field on the DSNTIPF installation panel. The value will be EBCDIC, ASCII, UNICODE, or 1-65533, and this session variable can never be null.

SYSIBM.COBOL_STRING_DELIMITER
Contains a string that corresponds to the value that is specified for the STRING DELIMITER field on the DSNTIPF installation panel. The value will be DEFAULT, '', or ', and this session variable can never be null.

SYSIBM.DATA_SHARING_GROUP_NAME
Contains a string that corresponds to the name of the data sharing group for this DB2 subsystem. If the subsystem is not part of data sharing group, the null value is returned.

SYSIBM.DATE_FORMAT
Contains a string that corresponds to the value that is specified for the DATE FORMAT field on the DSNTIP4 installation panel. The value will be ISO, JIS, USA, EUR, or LOCAL, and this session variable can never be null.

SYSIBM.DATE_LENGTH
Contains a string that corresponds to the value that is specified for the LOCAL DATE LENGTH field on the DSNTIP4 installation panel. The value will be 10-254, or 0 for no exit, and this session variable can never be null.

SYSIBM.DECIMAL_ARITHMETIC
Contains a string that corresponds to the value that is specified for the DECIMAL ARITHMETIC field on the DSNTIP4 installation panel. The value will be DEC15, DEC31, 15, or 31, and this session variable can never be null.

SYSIBM.DECIMAL_POINT
Contains a string that corresponds to the value that is specified for the DECIMAL POINT IS field on the DSNTIPF installation panel. The value will be '.' or ',' and this session variable can never be null.

SYSIBM.DEFAULT_DECFLOAT_ROUND_MODE
Contains a string that corresponds to the value that is specified for the DECFLOAT ROUNDING MODE field on the DSNTIPF installation panel. This session variable can never be null.

SYSIBM.DEFAULT_SSID
Contains a string that corresponds to the value that is specified for the GROUP ATTACH field on the DSNTIPK installation panel or the SUBSYSTEM NAME field on the DSNTIPM installation panel. This session variable can never be null.

SYSIBM.DEFAULT_LANGUAGE
Contains a string that corresponds to the value that is specified for the

LANGUAGE DEFAULT field on the DSNTIPF installation panel. The value will be ASM, C, CPP, IBMCOB, FORTRAN, or PL/I, and this session variable can never be null.

SYSIBM.DEFAULT_LOCALE_LC_CTYPE
Contains a string that corresponds to the value that is specified for the LOCALE LC_CTYPE field on the DSNTIPF installation panel. This session variable can never be null.

SYSIBM.DISTRIBUTED_SQL_STRING_DELIMITER
Contains a string that corresponds to the value that is specified for the DIST SQL STR DELIMTR field on the DSNTIPF installation panel. The value will be '', or ', and this session variable can never be null.

SYSIBM.DSNHDECP_NAME
Contains a string that corresponds to the fully qualified data set name of the data set from which the DSNHDECP or a user-specified application defaults module was loaded. For instance, 'DSN910.SDSNEXIT(DSNHDECP)'. This session variable can never be null.

SYSIBM.DYNAMIC_RULES
Contains a string that corresponds to the value that is specified for the USE FOR DYNAMICRULES field on the DSNTIP4 installation panel. The value will be YES or NO, and this session variable can never be null.

SYSIBM.ENCODING_SCHEME
Contains a string that corresponds to the value that is specified for the DEF ENCODING SCHEME field on the DSNTIPF installation panel. The value will be EBCDIC, ASCII, or UNICODE, and this session variable can never be null.

SYSIBM.MIXED_DATA
Contains a string that corresponds to the value that is specified for the MIXED DATA field on the DSNTIPF installation panel. The value will be YES or NO, and this session variable can never be null.

SYSIBM.NEWFUN
Contains a string that corresponds to the value that is specified for the INSTALL TYPE field on the DSNTIPA1 installation panel. The value will be INSTALL, UPDATE, MIGRATE, or ENFM, and this session variable can never be null. The value reflects the setting of the DSNHDECP variable NEWFUN.

SYSIBM.PACKAGE_NAME
Contains a string that corresponds to the name of the package that is currently being executed. If a package is not currently being executed, the null value is returned. (This situation can occur when the plan that is being executed bound one or more DBRMs directly).

SYSIBM.PACKAGE_SCHEMA
Contains a string that corresponds to the collection id of the package that is currently being executed. If a package is not currently being executed, the null value is returned.

SYSIBM.PACKAGE_VERSION
Contains a string that corresponds to the version of the package that is currently being executed. If a package is not currently being executed, the null value is returned.

SYSIBM.PAD_NUL_TERMINATED
Contains a string that corresponds to the value that is specified for the

PAD NUL-TERMINATED field on the DSNTIP4 installation panel. The value will be YES or NO, and this session variable can never be null.

SYSIBM.PLAN_NAME
Contains a string that corresponds to the name to the plan that is currently being executed. This session variable can never be null.

SYSIBM.SECLABEL
Contains a string that corresponds to the RACF SECLABEL value, if any, that has been defined for the current userid. If a value has not been defined, the null value is returned.

SYSIBM.SQL_STRING_DELIMITER
Contains a string that corresponds to the value that is specified for the SQL STRING DELIMITER field on the DSNTIPF installation panel. The value will be DEFAULT, '', or ', and this session variable can never be null.

SYSIBM.SSID
Contains a string that corresponds to the actual DB2 subsystem identifier for this DB2 subsystem. This session variable can never be null.

SYSIBM.STANDARD_SQL
Contains a string that corresponds to the value that is specified for the STD SQL LANGUAGE field on the DSNTIP4 installation panel. The value will be YES or NO, and this session variable can never be null.

SYSIBM.SYSTEM_NAME
Contains a string that corresponds to the name of the DB2 for z/OS subsystem, as defined in field SUBSYSTEM NAME on installation panel DSNTIPM. This session variable can never be null.

SYSIBM.SYSTEM_ASCII_CCSID
Contains a value that represents the ASCII CCSIDs that are in use on this system. The information is returned as a comma-delimited string that corresponds to the ASCII CCSID that was specified on installation panel DSNTIPF. The three values that are returned correspond to the SBCS, MIXED, and graphic CCSID that are in use for ASCII data on this system. A value of 65534 for the MIXED or graphic CCSID indicates that this system does not support storing data in that CCSID. This session variable can never be null.

SYSIBM.SYSTEM_EBCDIC_CCSID
Contains a value that represents the EBCDIC CCSIDs that are in use on this system. The information is returned as a comma-delimited string that corresponds to the EBCDIC CCSID that was specified on installation panel DSNTIPF. The three values that are returned correspond to the SBCS, MIXED, and graphic CCSID that are in use for EBCDIC data on this system. A value of 65534 for the MIXED or graphic CCSID indicates that this system does not support storing data in that CCSID. This session variable can never be null.

SYSIBM.SYSTEM_UNICODE_CCSID
Contains a value that represents the Unicode CCSIDs that are in use on this system. The information is returned as a comma-delimited string that corresponds to the UNICODE CCSID that was specified on installation panel DSNTIPF. The three values that are returned correspond to the SBCS, MIXED, and graphic CCSID that are in use for Unicode data on this system. This session variable can never be null.

SYSIBM.TIME_FORMAT
Contains a string that corresponds to the value that is specified for the

TIME FORMAT field on the DSNTIP4 installation panel. The value will be ISO, JIS, USA, EUR, or LOCAL, and this session variable can never be null.

SYSIBM.TIME_LENGTH
Contains a string that corresponds to the value that is specified for the LOCAL TIME LENGTH field on the DSNTIP4 installation panel. The value will be 8-254 or 0 for no exit, and this session variable can never be null.

SYSIBM.VERSION
Contains a string that represents the version of DB2. This string has the form *pppvvrrm* where:
- *ppp* is a product string that is set to the value 'DSN'
- *vv* is a two-digit version identifier such as '11'
- *rr* is a two-digit release identifier such as '01'
- *m* is a one-digit modification level.
  - Values 0, 1, 2, 3, and 4 are reserved for modification levels in conversion and enabling-new-function mode from Version 10 (CM10, CM10*, ENFM10, and ENFM10*)
  - Values 5, 6, 7, 8, and 9 are for modification levels in new-function mode.

This session variable can never be null.

For example, the following statement sets the value of host variable *hv1* to the name of the plan that is currently being executed:

```
SET :hv1 = GETVARIABLE('SYSIBM.PLAN_NAME');
```

For more information about the GETVARIABLE function, see "GETVARIABLE" on page 477.

References to array variables

An array variable is a variable that is defined as a user-defined array type.

An array variable can be defined in one of the following ways:
- An SQL parameter that is defined using the CREATE FUNCTION (SQL scalar) or CREATE PROCEDURE (SQL native) statement.
- An SQL variable that is defined using the DECLARE clause of a compound statement.

An array variable can be referenced in the following contexts:
- An input argument to the NULL predicate.
- An input argument to the ARRAY_EXISTS predicate.
- An input argument to a built-in array scalar function (ARRAY_DELETE, ARRAY_FIRST, ARRAY_LAST, ARRAY_NEXT, ARRAY_PRIOR, or TRIM_ARRAY).
- An argument to UNNEST specification.
- The outer SELECT list of a fullselect that does not include a set operator, in the definition of a cursor that is not scrollable. In this case a FETCH statement for the cursor must specify an array variable as the target for the corresponding result column of the fullselect for the array variable.
- The outer select list of a SELECT INTO statement, when the target for the corresponding column of the result table of the fullselect is an array variable.
- The outer select list of a scalar fullselect, on the right side of a SET *assignment-statement* statement or an SQL PL *assignment-statement* statement, when the corresponding target of the assignment is an array variable.

- The source value for a VALUES INTO statement, when the target for value is an array variable.
- The target of an assignment from a FETCH statement, when the corresponding source data is an array value.
- The target of a SELECT INTO statement, when source data for the corresponding column of the result table is an array value.
- The target of an assignment for a SET *assignment-statement* statement or an SQL PL *assignment-statement* statement, when the corresponding source value is an array value.
- The target of a VALUES INTO statement, when the source data value is an array value.
- An argument to or from a routine (CALL statement or function invocation).
- The value that is returned in a RETURN statement of an SQL scalar function.
- An ORDER BY or GROUP BY clause of an outer fullselect.

An array variable can also be referenced in an array element specification. An element of a user-defined array type can be referenced anywhere that an expression that returns the same data type as an element of that array can be used.

Restriction: An array variable or an array element must not be referenced in an SQL statement, other than a CALL statement, after a connection at a remote server has been established. This restriction includes the case of an SQL statement that is executing at a remote server as a result of a three-part name or an alias that resolves to an object at a remote server.

Related reference:

"Array element specification" on page 278

Host structures in PL/I, C, and COBOL

A *host structure* is a PL/I structure, C structure, or COBOL group that is referred to in an SQL statement.

In Java and REXX, there is no equivalent to a host structure. Host structures are defined by statements of the host language, as explained in *DB2 Application Programming and SQL Guide*. As used here, the term *host structure* does not include an SQLCA or SQLDA.

The form of a host structure reference is identical to the form of a host variable reference. The reference :S1:S2 is a host structure reference if S1 names a host structure. If S1 designates a host structure, S2 must be a small integer variable or an array of small integer variables. S1 is the host structure and S2 is its indicator array.

A host structure can be referred to in any context where a list of host variables can be referenced. A host structure reference is equivalent to a reference to each of the host variables contained within the structure in the order which they are defined in the host language structure declaration. The *n*th variable of the indicator array is the indicator variable for the *n*th variable of the host structure.

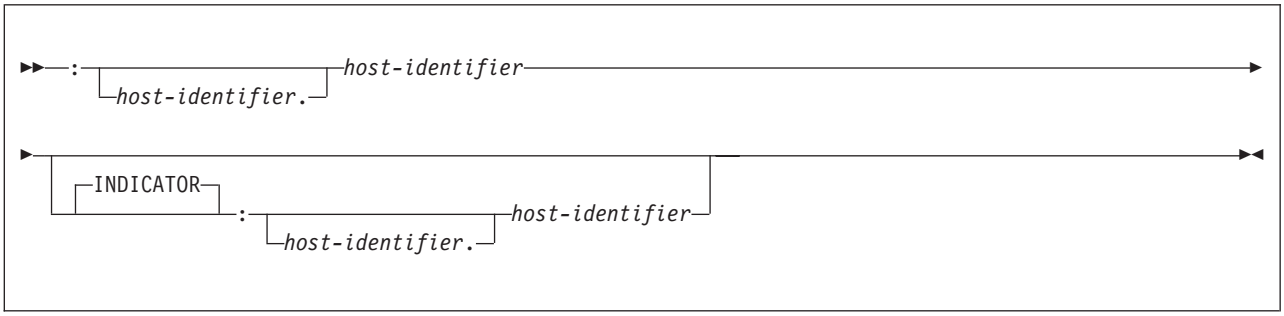
In PL/I, for example, if *V1*, *V2*, and *V3* are declared as the variables within the structure S1, the following two statements are equivalent:

```
EXEC SQL FETCH CURSOR1 INTO :S1;
EXEC SQL FETCH CURSOR1 INTO :V1, :V2, :V3;
```

If the host structure has *m* more variables than the indicator array, the last *m* variables of the host structure do not have indicator variables. If the host structure has *m* fewer variables than the indicator array, the last *m* variables of the indicator array are ignored. These rules also apply if a reference to a host structure includes an indicator variable or a reference to a host variable includes an indicator array. If an indicator array or variable is not specified, no variable of the host structure has an indicator variable.

In addition to structure references, individual host variables or indicator variables in PL/I, C, and COBOL can be referred to by qualified names. The qualified form is a host identifier followed by a period and another host identifier. The first host identifier must name a structure, and the second host identifier must name a host variable at the next level within that structure.

In PL/I, C, and COBOL, the syntax of *host-variable* is:



In general, a *host-variable* in an expression must identify a host variable (not a structure) described in the program according to the rules for declaring host variables. However, there are a few SQL statements that allow a host variable in an expression to identify a structure, as specifically noted in the descriptions of the statements.

The following examples show references to host variables and host structures:

```
:V1   :S1.V1   :S1.V1:V2   :S1.V2:S2.V4
```

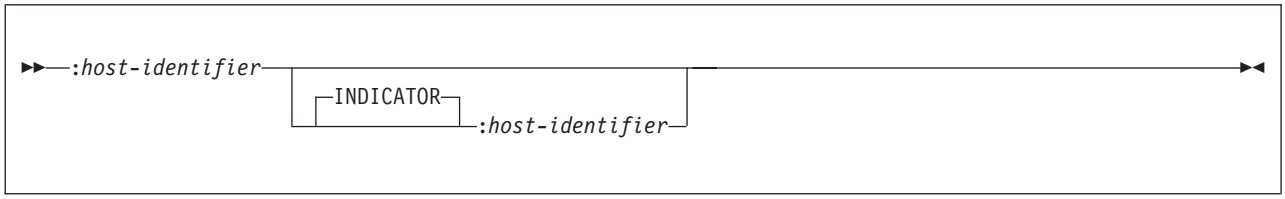
Host-variable-arrays in PL/I, C, C++, and COBOL

A *host-variable-array* is an array in which each element of the array contains a value for the same column. The first element in the array corresponds to the first value, the second element in the array corresponds to the second value, and so on. A host-variable-array can only be referenced in a FETCH statement for a multiple row fetch, in an INSERT statement with a multiple row insert, or in a multiple row MERGE statement.

Host-variable-arrays are defined by statements of the host language as explained in *DB2 Application Programming and SQL Guide*.

The form of a host structure reference is similar to the form of a host variable reference. The reference :COL1:COL1IND is a host-variable-array reference if COL1 designates an array. If COL1 designates an array, COL1IND must be a one dimensional array of small integer host variables. The dimension of the host-variable-array must be less than or equal to the dimension of the indicator array. If an indicator array is not specified, no variable of the main host-variable-array has an indicator variable.

In PL/I, C, C++, and COBOL, the syntax of *host-variable-array* is:



In the following example, *COL1* is the main host-variable-array and *COL1IND* is its indicator array, If *COL1* has 10 elements for fetching a single column of data for multiple rows of data, *COL1IND* must also have 10 entries.

```
EXEC SQL FETCH CURSOR FOR 5 ROWS  INTO :COL1 :COL1IND;
```

Functions

A *function* is an operation denoted by a function name followed by zero or more operands that are enclosed in parentheses.

It represents a relationship between a set of input values and a set of result values. The input values to a function are called *arguments*. For example, a function can be passed with two input arguments that have date and time data types and return a value with a timestamp data type as the result.

Types of functions

There are several ways to classify functions. One way to classify functions is as built-in functions, user-defined functions, or cast functions that are generated for distinct types.

Built-in functions

Built-in functions are functions that come with DB2 for z/OS. These functions provide a single-value result.

Built-in functions include operator functions such as "+", aggregate functions such as AVG, and scalar functions such as SUBSTR. For a list of the built-in aggregate and scalar functions and information on these functions, see Chapter 3, “Functions,” on page 337.

The built-in functions are in schema SYSIBM.

The RANK, DENSE_RANK, and ROW_NUMBER specifications are sometimes referred to as built-in functions. Refer to “OLAP specification” on page 282 for more information on these specifications.

User-defined functions

User-defined functions are functions that are created using the CREATE FUNCTION statement and registered to the DB2 in the catalog. These functions allow users to extend the function of DB2 by adding their own or third party vendor function definitions.

A user-defined function is an SQL, external, or sourced function. An SQL function is defined to the database using only SQL statements. An external function is defined to the database with a reference to an external program that is executed when the function is invoked. A sourced function is defined to the database with a

reference to a built-in function or another user-defined function. Sourced functions can be used to extend built-in aggregate and scalar functions for use on distinct types.

A user-defined function resides in the schema in which it was registered. The schema cannot be SYSIBM.

To help you define and implement user-defined functions, sample user-defined functions are supplied with DB2. You can also use these sample user-defined functions in your application program just as you would any other user-defined function if the appropriate installation job has been run. For a list of the sample user-defined functions, see "Sample user-defined functions" on page 2607. For more information on creating and using user-defined functions, see *DB2 Application Programming and SQL Guide*.

Generated user-defined functions for distinct types

Generated user-defined functions for distinct types (also called *cast functions*) are functions that DB2 automatically generates when a distinct type is created using the CREATE TYPE statement.

Cast functions support casting from the distinct type to the source type and from the source type to the distinct type. The ability to cast between the data types is important because a distinct type is compatible only with itself.

The generated cast functions reside in the same schema as the distinct type for which they were created. The schema cannot be SYSIBM.

Related reference:

"CREATE TYPE" on page 1510

Additional way to classify functions

Another way to classify functions is as aggregate, scalar, or table functions, depending on the input data values and result values.

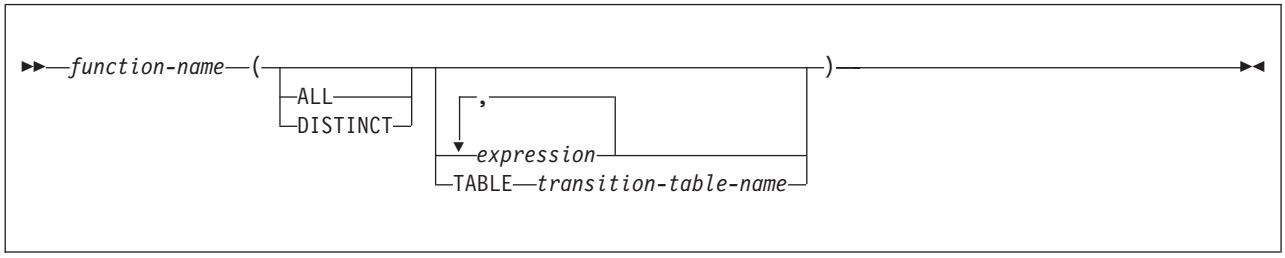
- An *aggregate function* receives a set of values for each argument (such as the values of a column) and returns a single-value result for the set of input values. Aggregate functions are sometimes called *column functions.* Built-in functions and user-defined sourced functions can be aggregate functions. Aggregate functions cannot be external user-defined function or SQL functions.
- A *scalar function* receives a single value for each argument and returns a single-value result. Built-in functions and user-defined functions, external, sourced, and SQL, can be scalar functions. The functions that are created for distinct types are also scalar functions.
- A *table function* returns a table for the set of arguments it receives. Each argument is a single value. A table function can only be referenced in the FROM clause of a subselect. A table function can be defined as an external or SQL function, but a table function cannot be a sourced function.

  Table functions can be used to apply SQL language processing power to data that is not stored in the database or to allow access to such data as if it were stored in a table. For example, a table function can read a file or get data from the web and return a result table.

For a list of the aggregate, scalar, and table functions and information on these functions, see Chapter 3, "Functions," on page 337.

Function invocation

Each reference to a scalar or aggregate function (either built-in or user-defined) conforms to the following syntax:



In the above syntax, *expression* cannot include an aggregate function. See "Expressions" on page 240 for other rules for *expression*.

The **ALL** or **DISTINCT** keyword can only be specified for an aggregate function or a user-defined function that is sourced on an aggregate function.

When a function is invoked within a trigger body, the **TABLE** keyword can be specified to indicate that an argument is a trigger transition table. In this case, the corresponding parameter of the function must have been defined with the **TABLE LIKE** clause.

Table functions can be referenced only in the FROM clause of a subselect. For more information on referencing a table function, see the description of the "from-clause" on page 773.

An array can only be specified as an argument to a function for a parameter that is defined with an array type. An array element specifies a scalar value, and can therefore be specified as an argument to a function when the data type of the array element is promotable to the data type of the corresponding parameter of the function definition. An argument that is an array can be specified only if the function is invoked from an SQL PL context. A function that returns an array can be invoked only from an SQL PL context.

When the function is invoked, the value of each of its parameters is assigned using storage assignment, to the corresponding parameter of the function. Control is passed to external functions according to the calling conventions of the host language. When execution of a user-defined aggregate or scalar function is complete, the result of the function is assigned, using storage assignment, to the result data type. For information about assignment rules, see "Assignment and comparison" on page 121.

Additionally, a character FOR BIT DATA argument cannot be passed as input for a parameter that is not defined as character FOR BIT DATA. Likewise, a character argument that is not FOR BIT DATA cannot be passed as input for a parameter defined as character FOR BIT DATA.

For compatibility with other SQL implementations, **UNIQUE** can be specified as a synonym for **DISTINCT** in aggregate functions.

Function resolution

After a function is invoked, DB2 must determine which function to execute. This process is called *function resolution* and it applies to both built-in and user-defined function.

A function is invoked by its function name, which is implicitly or explicitly qualified with a schema name, followed by parentheses that enclose the arguments to the function. Within the database, each function is uniquely identified by its *function signature*, which is its schema name, function name, the number of parameters, and the data types of the parameters. Thus, a schema can contain several functions that have the same name but each of which have a different number of parameters or parameters with different data types. Also, a function with the same name, number of parameters, and types of parameters can exist in multiple schemas.

Function resolution has two steps:

1. DB2 determines the set of candidate functions based on the qualification of the name of the invoked function, the unqualified name of the invoked function, and the number of arguments that are specified.
2. DB2 determines the best fit from the set of candidate functions based on the data types of the arguments of the invoked function as compared with the data types of the parameters of the functions in the set of candidate functions.

Function resolution is similar for functions that are invoked with a qualified or unqualified function name with the exception that for an unqualified name, DB2 needs to search more than one schema.

To improve performance of function resolution and to prevent potential issues as new functions are added, consider invoking user-defined functions by using a fully qualified name, including the schema name.

For a function invocation that passes a transition table, the data type, length, precision, and scale of each column in the transition table must exactly match the data type, length, precision, and scale of each column of the table that is named in the function definition.

The timestamp for the creation of a user-defined function must be older than the timestamp that results from an explicit bind for the plan or package that contains the function invocation. During autobind, built-in functions that are introduced in a DB2 release that is later than the DB2 release that is used to explicitly bind the package or plan are not considered for function resolution.

In a CREATE VIEW statement, function resolution occurs at the time the view is created. If another function with the same name is subsequently created, the view is not affected, even if the new function is a better fit than the one that was chosen at the time the view was created.

Qualified function resolution: When a function is invoked with a schema name and a function name, DB2 only searches the specified schema to resolve which function to execute.

DB2 selects candidate functions based on the following criteria:

- The name of the function instance must match the name in the function invocation.

- The number of input parameters in the function instance must match the number of arguments in the function invocation.
- The authorization ID of the statement must have the EXECUTE privilege to the function instance.

If no function meets these criteria, an error is returned. If one or more candidate functions are found in the schema, this set of candidate functions is processed for best fit.

For a function invocation that contains untyped parameter markers, the data types of those parameter markers are considered to match or be promotable to the data types of the parameters in the function instance.

Unqualified function resolution: When a function is invoked without a qualifier, DB2 searches the list of schemas in the SQL path to resolve which function instance to execute. For each schema in the SQL path, DB2 searches the schema for candidate functions based on the following criteria:

- The name of the function instance must match the name in the function invocation.
- The number of input parameters in the function instance must match the number of function arguments in the function invocation.
- The authorization ID of the statement must have the EXECUTE privilege on the function instance.

If DB2 does not find any candidate functions, an error is returned.

If no function meets these criteria, an error is returned. If one or more candidate functions are found in the schema, this set of candidate functions is processed for best fit.

For a function invocation that contains untyped parameter markers, the data types of those parameter markers are considered to match or be promotable to the data types of the parameters in the function instance.

Determining the best fit

More than one function with the same name might exist that is a candidate for execution. In that case, DB2 determines which function is the best fit for the invocation by comparing the data types of the parameters of each function in the set of candidate functions to determine which function satisfies the best fit requirements.

DB2 determines the function, or set of functions, that meet the best fit requirements for the invocation by comparing the argument and parameter data types. The data type of the result of the function or the type of function (aggregate, scalar, or table) under consideration does not enter into the determination of best fit.

When determining whether the data types of the parameters are the same as the arguments:

- Synonyms of data types match. For example, DOUBLE and FLOAT are considered to be the same.
- Attributes of a data type (such as length, precision, scale, CCSID) are ignored. Therefore, CHAR(8) and CHAR(35) are considered to be the same, as are DECIMAL(11,2) and DECIMAL(4,3).

- The character and graphic types are considered to be the same. For example, the following data types are considered to be the same type: CHAR and GRAPHIC, VARCHAR and VARGRAPHIC, and CLOB and DBCLOB. CHAR(13) and GRAPHIC(8) are considered to be the same type.
- For this argument, if one function has a data type that fits the function invocation better than the data types in the other functions, that function is the best fit. The precedence list for the promotion of data types in shows the data types that fit each data type, in best-to-worst order.
- If the data types of the first parameter for all the candidate functions fit the function invocation equally well, DB2 repeats this process for the next argument of the function invocation. DB2 continues this process for each argument until a best fit is found.

A subset of the candidate functions is obtained by considering only those functions for which the data type of each input argument of the function invocation matches or is promotable to the data type of the corresponding parameter of the function instance. The precedence list for the promotion of data types in Table 14 on page 110 shows the data types that fit (considering promotion) for each data type in best-to-worst order. If this subset is not empty, the best fit is determined using the promotable process on this subset of candidate functions. If this subset is empty, and the original set of candidate functions consisted of a single function, the best fit is determined using the castable process on the original candidate function. Otherwise, an error is returned.

Promotable process:

The *promotable process* determines the best fit for function resolution by considering only whether input arguments in the function invocation match or can be promoted to the data type of the corresponding parameter of the function definition.

For the subset of candidate functions, DB2 compares the parameter lists from left to right, using the following process:

- The data type of the argument in the function invocation is compared to the data type of the corresponding parameter in the definition of each candidate function. (synonyms of data types match and attributes of data type are ignored).
  - Attributes of a data type (such as length, precision, scale, CCSID) are ignored. Therefore, CHAR(8) and CHAR(35) are considered to be the same, as are DECIMAL(11,2) and DECIMAL(4,3).
  - The character and graphic types are considered to be the same. For example, the following data types are considered to be the same type: CHAR and GRAPHIC, VARCHAR and VARGRAPHIC, and CLOB and DBCLOB. CHAR(13) and GRAPHIC(8) are considered to be the same type.
- For this argument, if one candidate function has a data type that fits the function invocation better than the data types in the other candidate functions, that function is the best fit. The precedence list for the promotion of data types in Table 14 on page 110 shows the data types that fit each data type, in best-to-worst order.
- If the data types of the first parameter for more than one candidate functions fits the function invocation equally well, DB2 repeats this process for the next argument of the function invocation. DB2 continues this process for each argument until a best fit is found.

If only one candidate function remains after comparing all the arguments, that function is the best fit. If more than one candidate function remains, all the remaining candidate functions are considered to be equally the best fit. In this case, DB2 selects the function whose schema is first in the SQL path.

If a function is selected, its successful use depends on it being invoked in a context in which the returned result is allowed. For example, if the function returns a table where a table is not allowed, an error is returned.

Function resolution and input argument casting:

In considering the best fit of a candidate function, DB2 determines if the input arguments can be implicitly cast to the data type of the corresponding parameter for function resolution.

The *castable process* determines the best fit of a function, first considering if the input arguments in the function invocation match or can be promoted to the data type of the corresponding parameter of the function definition, and then if the input arguments can be implicitly cast to the data type of the corresponding parameter for function resolution. For the set of candidate functions, DB2 compares the parameter lists from left to right, using the following process:

- The data type of the argument in the function invocation is compared to the data type of the corresponding parameter in the definition of the candidate function to ensure that each argument can be promoted or cast to the corresponding parameter. If not, an error is returned.

If a function is selected, its successful use depends on it being invoked in a context in which the returned result is allowed. For example, if the function returns a table where a table is not allowed, an error is returned.

Implicit casting for function resolution: Implicit casting for function resolution is not supported for arguments with a user-defined type, binary, ROWID, or XML data type. It is also not supported for built-in or user-defined cast functions. Implicit casting is supported for the following cases:

- A numeric data type can be cast to a value of another numeric data type that is not in the data type promotion list for the source data type. This includes casting a numeric value to a numeric data type that is lower in the promotion list.
- A numeric data type can be cast to a character or graphic string data type, except for a LOB.
- A character or graphic string data type, except for a LOB, can be cast to a numeric data type.
- A character or graphic string data type, except for a LOB, can be cast to a date, time, or timestamp data type.
- A varying length character string data type, except for a LOB, can be cast to a fixed length character data type.

Best-fit consideration:

After determining the function that is the best fit, use of the function still might not be permitted. Each function is defined to return a result with a specific data type. If this result data type is not compatible with the context in which the function is invoked, an error occurs.

For example, assume functions named STEP are defined with different data types:

```
STEP(SMALLINT)returns CHAR(5)
STEP(DOUBLE)returns INTEGER
```

Assume also that the function is invoked with the following function reference (where S is a SMALLINT column):

```
SELECT ... 3+STEP(S) ...
```

Because there is an exact match on argument type, the first STEP is chosen. An error occurs on the statement because the result type is CHAR(5) instead of a numeric type as required for an argument of the addition operator.

In cases where the arguments of the function invocation are not an exact match to the data types of the parameters of the selected function, the arguments are converted to the data type of the parameter at execution using the same rules as assignment to columns. See "Assignment and comparison" on page 121. Problems with conversions can also occur when precision, scale, length, or the encoding scheme differs between the argument and the parameter. Conversion might occur for a character string argument when the corresponding parameter of the function has a different encoding scheme or CCSID. For example, an error occurs on function invocation when mixed data that actually contains DBCS characters is specified as an argument and the corresponding parameter of the function is declared with an SBCS subtype.

Additionally, a character FOR BIT DATA argument cannot be passed as input for a parameter that is not defined as character FOR BIT DATA. Likewise, a character argument that is not FOR BIT DATA cannot be passed as input for a parameter that is defined as character FOR BIT DATA.

An error also occurs in the following examples:

- The function is referenced in a FROM clause, but the function selected by the function resolution step is a scalar or aggregate function.
- The function calls for a scalar or aggregate function, but the function selected by the resolution step is a table function.

SQL path considerations for built-in functions

Function resolution applies to all functions, including built-in functions and other functions provided by DB2. If a function is invoked without its schema name, the SQL path is searched.

With the exception of the DB2 MQSeries® functions, the built-in functions are in schema SYSIBM.

Additional functions are available in other schemas, but are not considered as built-in functions because they are developed as user-defined functions that have no special processing considerations. User-defined functions cannot be defined in the SYSIBM schema (or any schema where the name begins with "SYS").

If SYSIBM is not first in the path, it is possible that DB2 will select another function instead of the intended built-in function. If schema "SYSIBM", "SYSFUN", "SYSPROC", "SYSIBMADM" is not explicitly specified in the SQL path, the schema is implicitly assumed at the front of the path. DB2 adds implicitly assumed schemas in the order of "SYSIBM", "SYSFUN", "SYSPROC", "SYSIBMADM".

Related concepts:

"SQL path" on page 64

Version resolution

Normally, the currently active version of an SQL function is used for invocation of the function.

However, if the invocation is a recursive invocation that occurs inside the body of the same function, and the currently active version has changed since the original invocation, the active version is not used. The version that is used in the original invocation is used for any recursive invocation until the entire function completes. This preserves the semantics of the version that is used by the original invocation.

The version used in the original invocation is also used when the recursive invocation is indirect. For example, assume that function FN1 invokes function FN2, which in turn invokes FN1 (indirect, recursive invocation). The invocation of function FN1 in function FN2 uses the version of FN1 that is active at the time of the original invocation of function FN1.

Since the currently active version is used at the next invocation (except in recursive invocations), it is possible that two or more versions of the same function can be run by a given thread. For example, an invocation of function FN1 in an application causes the currently active version of FN1 to load and execute. During or after execution of the original invocation of FN1, an ALTER FUNCTION statement that specifics ACTIVE VERSION FN1_V2 is run and changes the active version of the function FN1 to version FN1_V2. Subsequent invocations of function FN1 from the same thread will load and execute the currently active version of the function, FN1_V2.

Examples of function resolution

The following examples illustrate function resolution.

Example 1: Assume that MYSCHEMA contains two functions, both named FUNA, that were registered with these partial CREATE FUNCTION statements.

```
1.  CREATE FUNCTION MYSCHEMA.FUNA (VARCHAR(10), INT, DOUBLE) ...
2.  CREATE FUNCTION MYSCHEMA.FUNA (VARCHAR(10), REAL, DOUBLE) ...
```

Also assume that a function with three arguments of data types VARCHAR(10), SMALLINT, and DECIMAL is invoked with a qualified name:

```
MYSCHEMA.FUNA(VARCHARCOL, SMALLINTCOL, DECIMALCOL)
```

Both MYSCHEMA.FUNA functions are candidates for this function invocation because they meet the criteria specified in "Function resolution" on page 234. The data types of the first parameter for the two function instances in the schema, which are both VARCHAR, fit the data type of the first argument of the function invocation, which is VARCHAR, equally well. However, for the second parameter, the data type of the first function (INT) fits the data type of the second argument (SMALLINT) better than the data type of second function (REAL). Therefore, DB2 selects the first MYSCHEMA.FUNA function as the function instance to execute.

Example 2: Assume that these functions were registered with these partial CREATE FUNCTION statements:

```
1.  CREATE FUNCTION SMITH.ADDIT (CHAR(5), INT, DOUBLE) ...
2.  CREATE FUNCTION SMITH.ADDIT (INT, INT, DOUBLE) ...
3.  CREATE FUNCTION SMITH.ADDIT (INT, INT, DOUBLE, INT) ...
4.  CREATE FUNCTION JOHNSON.ADDIT (INT, DOUBLE, DOUBLE) ...
```

```
5.  CREATE FUNCTION JOHNSON.ADDIT (INT, INT, DOUBLE) ...
6.  CREATE FUNCTION TODD.ADDIT (REAL) ...
7.  CREATE FUNCTION TAYLOR.SUBIT (INT, INT, DECIMAL) ...
```

Also assume that the SQL path at the time an application invokes a function is "TAYLOR" "JOHNSON", "SMITH". The function is invoked with three data types (INT, INT, DECIMAL) as follows:

```
SELECT ... ADDIT(INTCOL1, INTCOL2, DECIMALCOL) ...
```

Function 5 is chosen as the function instance to execute based on the following evaluation:

- Function 6 is eliminated as a candidate because schema TODD is not in the SQL path.
- Function 7 in schema TAYLOR is eliminated as a candidate because it does not have the correct function name.
- Function 1 in schema SMITH is eliminated as a candidate because the INT data type is not promotable to the CHAR data type of the first parameter of Function 1.
- Function 3 in schema SMITH is eliminated as a candidate because it has the wrong number of parameters.
- Function 2 is a candidate because the data types of its parameters match or are promotable to the data types of the arguments.
- Both Function 4 and 5 in schema JOHNSON are candidates because the data types of their parameters match or are promotable to the data types of the arguments. However, Function 5 is chosen as the better candidate because although the data types of the first parameter of both functions (INT) match the first argument (INT), the data type of the second parameter of Function 5 (INT) is a better match of the second argument (INT) than Function 4 (DOUBLE).
- Of the remaining candidates, Function 2 and 5, DB2 selects Function 5 because schema JOHNSON comes before schema SMITH in the SQL path.

Expressions

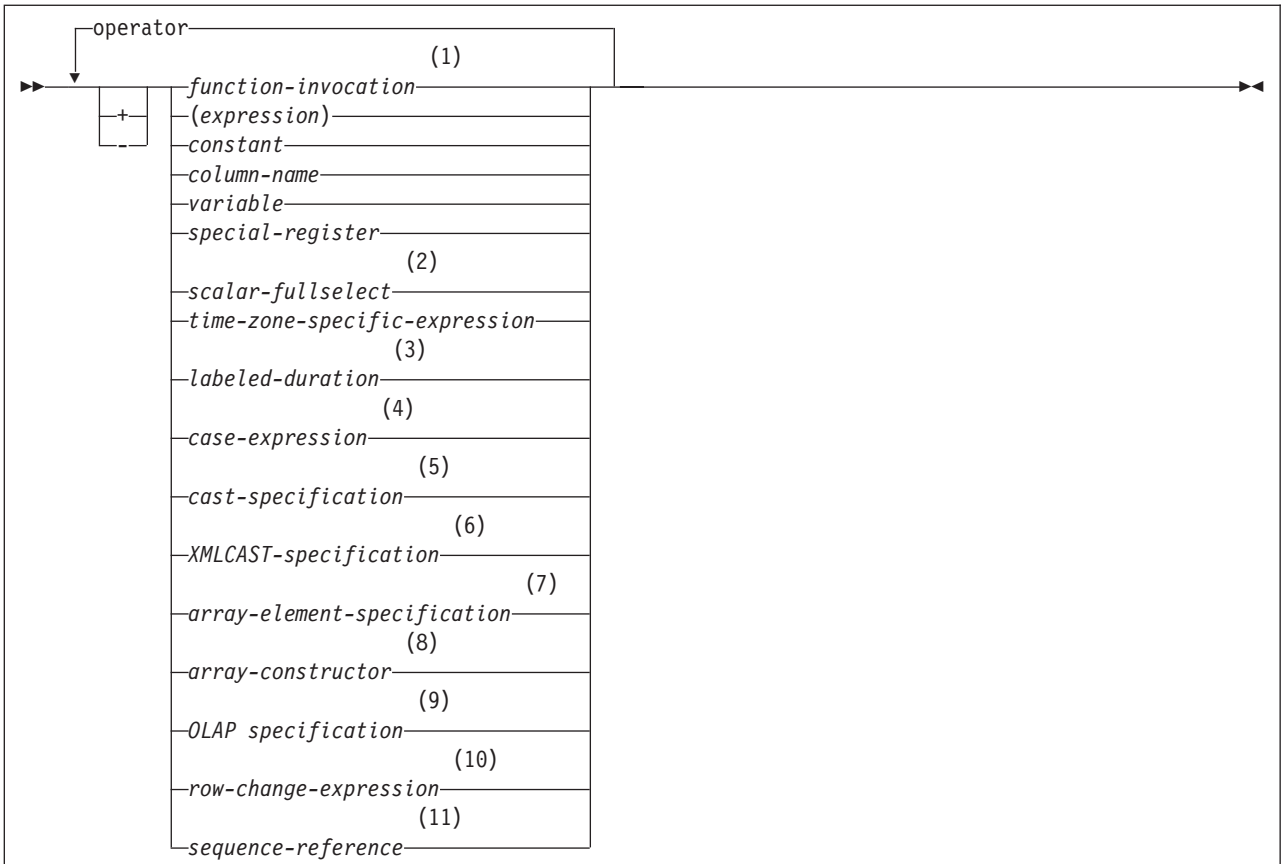
An *expression* specifies a value and can take a number of different forms.

Authorization: The use of some of the expressions, such as a *scalar-fullselect*, *sequence-reference*, *global-variable*, or *function-invocation*, requires having the appropriate authorization. For these objects, the privilege set that is defined below must include the following authorization:

- *cast-specification*. The authorization to reference a user-defined type in a cast specification. For information about authorization considerations, see "CAST specification" on page 267.
- *function-invocation*. Authorization to execute the function. For information about how the particular function is chosen and authorization considerations, see "Function resolution" on page 234.
- *scalar-fullselect*. For information about authorization considerations, see "Authorization" on page 762.
- *sequence-reference*. The USAGE privilege on the specified sequence, ownership of the sequence, DATAACCESS authority, or SYSADM authority. For example, with a sequence reference, USAGE authorization on the sequence is required.
- *global-variable*. The READ privilege on the specified global variable, ownership of the global variable, DATAACCESS authority, or SYSADM authority.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set is the union of the privilege sets that are held by each authorization ID of the process.

The form of an expression is as follows:



Notes:

1 Must be a scalar function. See "Functions" on page 231 for more information.

2 See "Scalar-fullselect" on page 253 for more information.

3 See Labeled durations for more information.

4 See "CASE expressions" on page 263 for more information.

5 See "CAST specification" on page 267 for more information.

6 See "XMLCAST specification" on page 276 for more information.

7 See "Array element specification" on page 278 for more information.

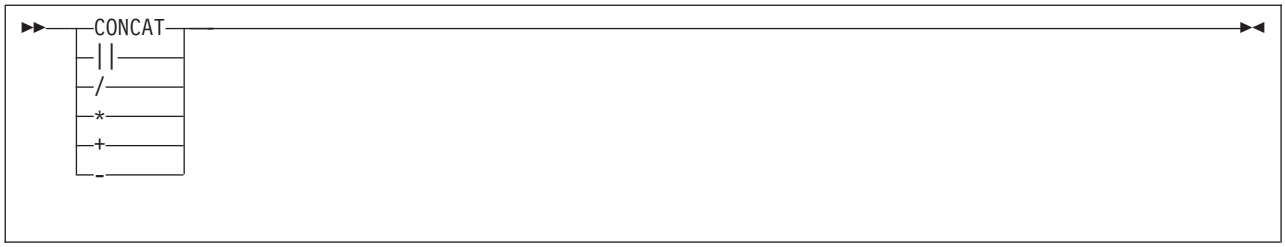
8 See "Array constructor" on page 280 for more information.

9 See "OLAP specification" on page 282

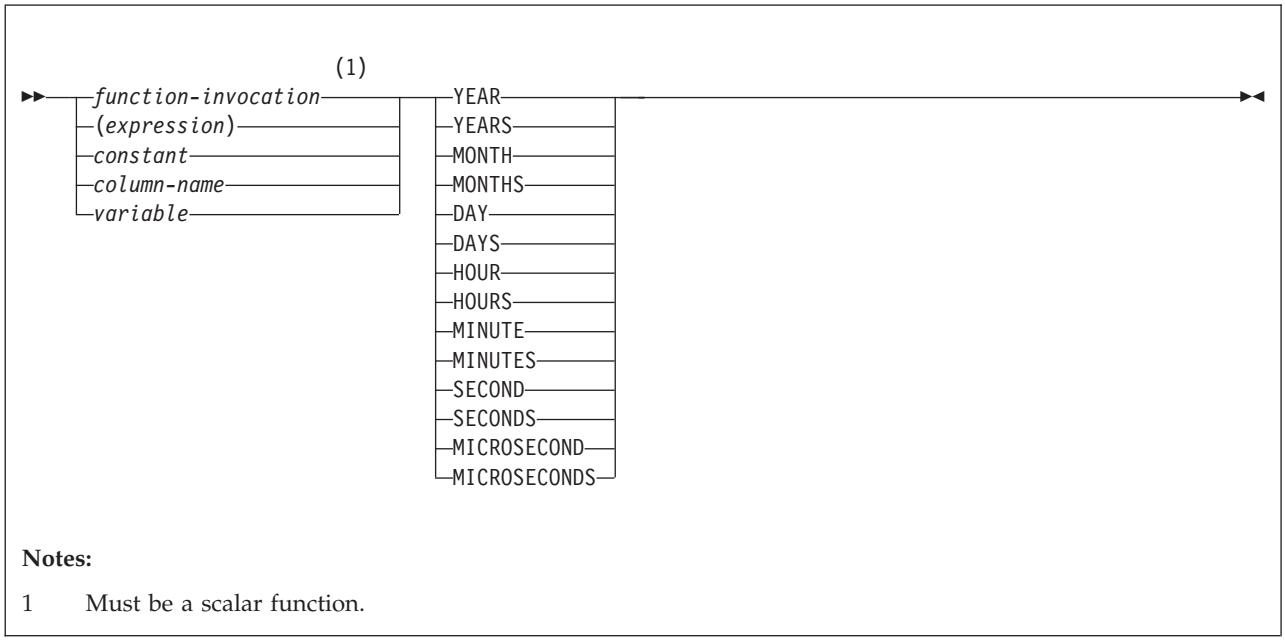
10 See "ROW CHANGE expression" on page 289

11 See "Sequence reference" on page 291

operator:



labeled-duration:



Notes:

1 Must be a scalar function.

Expressions without operators

If no operators are used, the result of the expression is the specified value.

Examples:

```
SALARY     :SALARY     'SALARY'    MAX(SALARY)
```

Expressions with arithmetic operators

If arithmetic operators are used, the result of the expression is a number derived from the application of the operators to the values of the operands.

The result of the expression can be null. If any operand has the null value, the result of the expression is the null value. Arithmetic operators (except unary plus, which is meaningless) must not be applied to strings. For example, USER+2 is invalid. Multiplication and division operators must not be applied to datetime values, which can only be added and subtracted.

The prefix operator + (*unary plus*) does not change its operand. The prefix operator - (*unary minus*) reverses the sign of a nonzero operand. If the data type of *A* is *small integer*, the data type of *-A* is *large integer*. The first character of the token following a prefix operator must not be a plus or minus sign.

The *infix operators* +, -, *, and / specify addition, subtraction, multiplication, and division, respectively. The value of the second operand of division must not be zero.

Arithmetic with two integer operands

If both operands of an arithmetic operator are integers, the operation is performed in binary. The result is a large integer unless either (or both) operand is a big integer, in which case the result is a big integer.

The result of an integer arithmetic operation (including unary minus) must be within the range of the result type.

Arithmetic with an integer and a decimal operand

If one operand is an integer and the other operand is decimal, the operation is performed in decimal. The arithmetic operation uses a temporary copy of the integer that has been converted to a decimal number.

The temporary copy of the integer that has been converted to a decimal number has a precision *p* and scale 0. *p* is 19 for a big integer, 11 for a large integer, and 5 for a small integer. In the case of an integer constant, *p* depends on the number of digits in the integer constant. *p* is 5 for an integer constant consisting of 5 digits or fewer. Otherwise, *p* is the same as the number of digits in the integer constant.

Arithmetic with an integer and a decimal floating-point operand

If one operand is a small integer, large integer, or big integer and the other is a decimal floating-point number, the operation is performed in decimal floating point. The arithmetic operation uses a temporary copy of the integer that has been converted to a decimal floating-point number.

For small integer or large integer, the temporary copy of the integer is converted to DECFLOAT(16). For big integer, the temporary copy of the big integer is converted to DECFLOAT(34). The rules for two decimal floating point operands are then applied.

Arithmetic with two decimal operands

If both operands are decimal, the operation is performed in decimal.

The result of any decimal arithmetic operation is a decimal number with a precision and scale that depend on two factors:

The precision and scale of the operands
: In the discussion of operations with two decimal operands, the precision and scale of the first operand are denoted by *p* and *s*, that of the second operand by *p'* and *s'*. Thus, for a division, the dividend has precision *p* and scale *s*, and the divisor has precision *p'* and scale *s'*.

Whether DEC31 or DEC15 is in effect for the operation
: DEC31 and DEC15 specify the rules to be used when both operands in a decimal operation have precisions of 15 or less. DEC15 specifies the rules which do not allow a precision greater than 15 digits, and DEC31 specifies the rules which allow a precision of up to 31 digits. The rules for DEC31 are always used if either operand has a precision greater than 15.

For static SQL statements, the value of the field DECIMAL ARITHMETIC on installation panel DSNTIP4 or the SQL processing option DEC determines whether DEC15 or DEC31 is used.

For dynamic SQL statements, the value of the field DECIMAL ARITHMETIC on installation panel DSNTIP4, the SQL processing option DEC, or the special register CURRENT PRECISION determines whether DEC15 or DEC31 is used according to these rules:

- Field DECIMAL ARITHMETIC applies if either of these conditions is true:
  - DYNAMICRULES run behavior applies and the application has not set CURRENT PRECISION.

    For a list of the DYNAMICRULES option values that specify run, bind, define, or invoke behavior, see Table 6 on page 75.
  - DYNAMICRULES bind, define, or invoke behavior applies; the value of installation panel field USE FOR DYNAMICRULES is YES; and the application has not set CURRENT PRECISION.
- SQL processing option DEC applies if DYNAMICRULES bind, define, or invoke behavior is in effect, the value of installation panel field USE FOR DYNAMICRULES is NO, and the application has not set CURRENT PRECISION.
- Special register CURRENT PRECISION applies if the application sets the register.

The value of DECIMAL ARITHMETIC is the default value for the SQL processing option and the special register. SQL statements executed using SPUFI use the value in DECIMAL ARITHMETIC.

Decimal addition and subtraction:

For decimal operations, the precision and scale of the result depends on the precision and scale of the operands.

If the operation is addition or subtraction and the operands do not have the same scale, the operation is performed with a temporary copy of one of the operands that has been extended with trailing zeros so that its fractional part has the same number of digits as the other operand.

The precision of the result is the minimum of *n* and the quantity `MAX(p-s,p'-s')+MAX(s,s')+1`. The scale is `MAX(s,s')`. *n* is 31 if DEC31 is in effect or if the precision of at least one operand is greater than 15. Otherwise, *n* is 15.

In COBOL, blanks must precede and follow a minus sign to avoid any ambiguity with COBOL host variable names (which allow the use of a dash).

Decimal multiplication:

For decimal multiplication, the precision and scale of the result depends on the precision and scale of the operands.

For multiplication, the precision of the result is `MIN(n,p+p')`, and the scale is `MIN(n,s+s')`. *n* is 31 if DEC31 is in effect or if the precision of at least one operand is greater than 15. Otherwise, *n* is 15.

If both operands have a precision greater than 15, the operation is performed using a temporary copy of the operand with the smaller precision. If the operands have the same precision, the second operand is selected. If more than 15 significant digits are needed for the integral part of the copy, the statement's execution is ended and an error occurs. Otherwise, the copy is converted to a number with precision 15, by truncating the copy on the right. The truncated copy has a scale of

MAX(0,*S*-(*P*-15)), where *P* and *S* are the original precision and scale. If, in the process of truncation, one or more nonzero digits are removed, SQLWARN7 in SQLCA is set to W, indicating loss of precision.

When both operands have a precision greater than 15, the foregoing formulas for the precision and scale of the result still apply, with one change: for the operand selected as the copy, use the precision and scale of the truncated copy; that is, use 15 as the precision and MAX(0,*S*-(*P*-15)) for the scale.

Let *n* denote the value of the operand with the greater precision or the first operand in the case of operands with the same precision. The number of leading zeros in a 31-digit representation of *n* must be greater than the precision of the other operand. This is always the case if the precision of the operand is 15 or less. With greater precisions, overflow can occur even if the precision of the result is less than 31. For example, the expression:

```
1000000000000000000000000. * 1
```

will cause overflow because the number of leading zeros in the 31-digit representation of the large number and the precision of the small number are both 5 (see "Arithmetic with an integer and a decimal operand" on page 244).

Decimal division:

The rules for a specific decimal division depend on whether the DEC31 option is in effect for the operation, whether p is greater than 15, and whether p' is greater than 15

The following table shows how the precision and scale of the result depend on these factors. In that table, the occurrence of "N/A" in a row implies that the indicated factor is not relevant to the case represented by the row.

Table 42. Precision (p) and scale (s) of the result of a decimal division

DEC31	*p*	*p'*	*P*	*S*
Not in effect	≤15	≤15	15	15-(*p*-*s*+*s'*)
In effect	≤15	≤15	31	*N*-(*p*-*s*+*s'*), where *N* is 30-*p'* if *p'* is odd. *N* is 29-*p'* if *p'* is even.
N/A	>15	≤15	31	N-(*p*-*s*+*s'*), where *N* is 30-*p'* if *p'* is odd. *N* is 29-*p'* if *p'* is even.
N/A	N/A	>15	31	15-(*p*-*s*+*x*), where *x* is MAX(0,*s'*-(*p'*-15)) (See the following note)

Notes on decimal division: If *p'* is greater than 15, the division is performed using a temporary copy of the divisor. If more than 15 significant digits are needed for the integral part of the divisor, the statement's execution is ended, and an error occurs. Otherwise, the copy is converted to a number with precision 15, by truncating the copy on the right. The truncated copy has a scale of MAX(0,*s'*-(*p'*-15)), which is the formula for *x*. If, in the process of truncation, one or more nonzero digits are removed, SQLWARN7 in SQLCA is set to W, indicating loss of precision.

Minimum divide result scale

If the calculated value of '*s*' is negative, an error occurs. If a minimum divide result scale is specified, this error does not occur.

The minimum scale is determined according to the following precedence:

Static SQL

1. The precompiler DEC option, if it is set with a non-zero scale.
2. The DECIMAL ARITHMETIC field (DECARTH) on installation panel DSNTIP4, if it is set with a non-zero scale.
3. The MINIMUM DIVIDE SCALE opaque subsystem parameter (MINDVSCL), if it is set to a value other than NONE.
4. The MINIMUM DIVIDE SCALE field (DECDIV3) on installation panel DSNTIP4, if it is set to YES.

Dynamic SQL

1. The CURRENT PRECISION special register, if it is set with a non-zero scale.
2. Either of the following cases:
   - For a package that was bound with DYNAMICRULES RUN or if the USE FOR DYNAMICRULES field (DYNRULS) on installation panel DSNTIP4 is set to YES: The DECIMAL ARITHMETIC field (DECARTH) on installation panel DSNTIP4, if it is set with a non-zero scale.
   - For all other cases: The precompiler DEC option, if it is set with a non-zero scale.
3. The MINIMUM DIVIDE SCALE opaque subsystem parameter (MINDVSCL), if it is set to value other than NONE.
4. The MINIMUM DIVIDE SCALE field (DECDIV3) on installation panel DSNTIP4, if it is set to YES

SQL statements that are executed using SPUFI

The value in DECIMAL ARITHMETIC (DECARTH).

The default value for both the precompiler DEC option and the CURRENT PRECISION special register is DECIMAL ARITHMETIC.

A minimum divide result scale of 3 can be specified using the MINIMUM DIVIDE SCALE field on the installation panel DSNTIP4. A minimum divide scale result between 1 and 9 can be specified using the DECIMAL ARITHMETIC OPTION of the form 'D*pp.s*' where '*pp*' is 15 or 31 and represents the precision and '*s*' represents the minimum divide scale, as a number between 1 and 9. Such a specification overrides the MINIMUM DIVIDE SCALE. When a minimum divide result scale is specified, the formula `MAX(s,s')`, where *s* represents the scale derived from the above table and *s'* represents the value specified by the minimum divide result scale, is applied and a new scale is derived. The newly derived scale is the scale of the result and overrides any scale derived using the table above.

Arithmetic with a decimal and a decimal floating-point operand

If one operand is a decimal and the other is a decimal floating point, the operation is performed in decimal floating point. The arithmetic operation uses a temporary copy of the decimal that has been converted to a decimal floating point based on the precision of the decimal number.

If the decimal number has a precision of less than 17, the decimal number is converted to DECFLOAT(16). Otherwise, the decimal number is converted to DECFLOAT(34). The rules for two decimal floating-point operands are then applied.

Arithmetic with floating-point operands

If either operand of an arithmetic operator is floating-point, the operation is performed in floating-point. If necessary, the operands are first converted to double-precision floating-point numbers. Thus, if any element of an expression is a floating-point number, the result of the expression is a double-precision floating-point number.

An operation involving a floating-point number and an integer is performed with a temporary copy of the integer that has been converted to double-precision floating-point. An operation involving a floating-point number and a decimal number is performed with a temporary copy of the decimal number that has been converted to double-precision floating-point. The result of a floating-point operation must be within the range of floating-point numbers.

The order in which floating-point operands (or arguments to functions) are processed can affect the results slightly because floating-point operands are approximate representations of real numbers. Because the order in which operands are processed might be implicitly modified by DB2 (for example, DB2 might decide what degree of parallelism to use and what access plan to use), an application that uses floating-point operands should not depend on the results being precisely the same each time an SQL statement is executed.

Arithmetic with a floating-point and a decimal floating-point operand

If one operand is a floating-point number (real or double) and the other is a decimal floating-point number, the operation is performed in decimal floating-point. The arithmetic operation uses a temporary copy of the floating-point number that has been converted to a decimal floating-point number.

Arithmetic with two decimal floating-point operands

If both operands are decimal floating point, the operation is performed in decimal floating point. If one operand is DECFLOAT(*n*) and the other is DECFLOAT(*m*), the operation is performed in DECFLOAT(max(*n*,*m*)).

General Arithmetic Operation Rules for DECFLOAT:

Certain general rules apply to all arithmetic operations on the DECFLOAT data type.

The following general rules apply to all arithmetic operations on the DECFLOAT data type:

- Every operation on finite numbers is carried out as though an exact mathematical result is computed, using integer arithmetic on the coefficient where possible.

  If the coefficient of the theoretical exact result has no more than the number of digits that reflect its precision (16 or 34), it is used for the result without change (unless there is an underflow or overflow condition). If the coefficient has more than the number of digits that reflect its precision, it is rounded to exactly the number of digits that reflect its precision (16 or 34), and the exponent is increased by the number of digits that are removed.

For static SQL statements other than CREATE VIEW, the ROUNDING bind option or the native SQL procedure option determines the rounding mode.

For dynamic SQL statements (and static CREATE VIEW statements), the special register CURRENT DECFLOAT ROUNDING MODE determines the rounding mode.

If the value of the adjusted exponent of the result is less than E_{min}, an exception condition is returned. In this case, the calculated coefficient and exponent form the result, unless the value of the exponent is less than E_{tiny}, in which case the exponent is set to E_{tiny}, the coefficient is rounded (possibly to zero) to match the adjustment of the exponent, and the sign is unchanged. If this rounding gives an inexact result, an underflow exception condition is returned.

If the value of the adjusted exponent of the result is larger than E_{max}, an overflow exception condition is returned. In this case, the result is as defined as an overflow exception condition and might be infinite. It will have the same sign as the theoretical result.

- Arithmetic that uses the special value infinity follows the usual rules, where negative infinity is less than every finite number and positive infinity is greater than every finite number.

  Under these rules, an infinite result is always exact. Certain uses of infinity return an invalid operation condition. The following list is a list of operations that can cause an invalid operation condition and the result of the operation is NaN when one of the operands is infinity but the other operand is not NaN nor sNaN.

  – Add +infinity to -infinity during an addition or subtraction operation
  – Multiply 0 by +infinity or -infinity
  – Divide either +infinity or -infinity by either +infinity or -infinity
  – The dividend for a MOD function is either +infinity or -infinity
  – Either argument of the QUANTIZE function is +infinity or -infinity
  – The second argument of the POWER® function is +infinity or -infinity
  – Signaling NaNs when used as an operand to an arithmetic operation

  The following arithmetic rules apply to arithmetic operations and the NaN value:

  – The result of any arithmetic operation which has an operand which is a NaN (a quiet NaN or signaling NaNs) is NaN. The sign of the result is copied from the first operand which is a signaling NaN, or if neither operand is signaling then the sign is copied from the first operand which is a NaN. Whenever a result is a NaN, the sign of the result depends only on the copied operand.
  – The sign of the result of a multiplication or division will be negative only if the operands have different signs and neither is a NaN.
  – The sign of the result of an addition or subtraction will be negative only if the result is less than zero and neither operand is a NaN, except for the following cases where the result is a negative 0:
    - A result is rounded to zero, and the value, before rounding, had a negative sign
    - Subtract 0 from -0
    - Addition of operands with opposite signs (or subtraction of operands with the same sign), the result has a coefficient of 0, and the rounding mode is ROUND_FLOOR
    - Multiplication or division and the result has a coefficient of 0 and the signs of the operands are different

- The first argument of the POWER function is -0, and the second argument is a positive odd number
- The argument of the CEIL, FLOOR, or SQRT function is -0
- The first argument of the ROUND or TRUNCATE function is -0

Examples involving special DECFLOAT values:

```
INFINITY + 1            = INFINITY
INFINITY + INFINITY     = INFINITY
INFINITY + -INFINITY    = NAN        -- exception
NAN + 1                 = NaN
NAN + INFINITY          = NaN
1 - INFINITY            = -INFINITY
INFINITY - INFINITY     = NAN        -- exception
-INFINITY - -INFINITY   = NAN        -- exception
-0.0 - 0.0E1            = -0
-1.0 * 0.0E1            = -0
1.0E1 / 0               = INFINITY
-1.0E5 / 0.0            = -INFINITY
1.0E5 / -0              = -INFINITY
INFINITY / -INFINITY    = NAN        -- exception
INFINITY / 0            = INFINITY
-INFINITY / 0           = -INFINITY
-INFINITY / -0          = INFINITY
```

Arithmetic with distinct type operands

A distinct type cannot be used with arithmetic operators even if its source data type is numeric.

To perform an arithmetic operation, create a function with the arithmetic operator as its source. For example, if there were distinct types INCOME and EXPENSES, both of which had DECIMAL(8,2) data types, the following user-defined function, REVENUE, could be used to subtract one from the other.

```
CREATE FUNCTION REVENUE ( INCOME, EXPENSES )
  RETURNS DECIMAL(8,2) SOURCE "-" ( DECIMAL, DECIMAL)
```

Alternately, the - (minus) operator could be overloaded using a function to subtract the new data types.

```
CREATE FUNCTION "-" ( INCOME, EXPENSES )
  RETURNS DECIMAL(8,2) SOURCE "-" ( DECIMAL, DECIMAL)
```

Alternatively, the distinct type can be cast to a built-in data type and the result used as an operand of an arithmetic operator.

Expressions with the concatenation operator

When two strings operands are concatenated, the result of the expression is a string.

The operands of concatenation must be compatible strings. A binary string cannot be concatenated with a character string, including character strings that are defined as FOR BIT DATA (for more information on the compatibility of data types, see the compatibility matrix in Table 23 on page 121). A distinct type that is based on a string type can be concatenated only if an appropriate user-defined function is created.

13. In various EBCDIC code pages, DB2 supports code point combinations X'4F4F', X'BBBB', and X'5A5A' to mean concatenation. X'BBBB' and X'5A5A' are interpreted to mean concatenation only on single byte character set DB2 subsystems.

Both CONCAT and the vertical bars (||) represent the concatenation operator. Vertical bars (or the characters that must be used in place of vertical bars in some countries[13]) can cause parsing errors in statements passed from one DBMS to another. The problem occurs if the statement undergoes character conversion with certain combinations of source and target CCSIDs[13]. Thus, CONCAT is the preferable concatenation operator.

If either operand can be null, the result can be null, and if either is null, the result is the null value. Otherwise, the result consists of the first operand string followed by the second.

The following table shows how the string operands determine the data type and the length attribute of the result (the order in which the operands are concatenated has no effect on the result).

Table 43. Data type and length of concatenated operands

If one operand column is	And the other operand is	The data type of the result column is[1]
CHAR(*x*)	CHAR(*y*) with a combined length attribute that is less than 256	CHAR(*x*+*y*)[2]
	CHAR(*y*) with a combined length attribute that is greater than 255	VARCHAR(MIN(*x*'+*y*',32764))[3]
	VARCHAR(*y*)	
VARCHAR(*x*)	VARCHAR(*y*)	VARCHAR(MIN(*x*'+*y*',32764))[3]
CLOB(*x*)	CHAR(*y*)	CLOB(MIN(*x*'+*y*',2G))
	VARCHAR(*y*)	
	CLOB(*y*)	
	GRAPHIC(*y*)	DBCLOB(MIN(*x*+*y*,1G))
	VARGRAPHIC(*y*)	
	DBCLOB(*y*)	
GRAPHIC(*x*)	CHAR(*y*)	VARGRAPHIC(MIN(*x*+*y*,16382)) [4]
	VARCHAR(*y*)	
	VARGRAPHIC(*y*)	
VARGRAPHIC(*x*)	CHAR(*y*)	VARGRAPHIC(MIN(*x*+*y*,16382)) [4]
	VARCHAR(*y*)	
	GRAPHIC(*y*)	
	GRAPHIC(*y*)	
DBCLOB(*x*)	CHAR(*y*)	DBCLOB(MIN(*x*+*y*,1G))
	VARCHAR(*y*)	
	CLOB(*y*)	
	GRAPHIC(*y*)	
	VARGRAPHIC(*y*)	
	DBCLOB(*y*)	

Table 43. Data type and length of concatenated operands (continued)

If one operand column is	And the other operand is	The data type of the result column is[1]
BINARY(*x*)	BINARY(*y*) with a combined length attribute that is less than 256	BINARY(*x*+*y*)
	BINARY(*y*) with a combined length attribute that is greater than 255	VARBINARY(MIN(*x*+*y*,32764))
VARBINARY(*x*)	VARBINARY(*y*)	VARBINARY(MIN(*x*+*y*,32764))
	BINARY(*y*)	
BLOB(*x*)	BLOB(*y*)	BLOB(MIN(*x*+*y*, 2G))

Notes:

1.
   - 2G represents 2,147,483,647 bytes
   - 1G represents 1,073,741,823 double-byte characters
2. Neither CHAR(*x*) nor CHAR(*y*) can contain mixed data. If either operand contains mixed data, the result is VARCHAR(MIN(*x*'+*y*',32764)).
3. If conversion of the first operand is required, *x*' = 3*x*; otherwise, it remains *x*. If conversion of the second operand is required, *y*'= 3*y*; otherwise, it remains *y*.
4. Both operands are converted to UTF-16, if necessary (that is, the operand is not already UTF-16), and the results are concatenated.

As the previous table shows, the length of the result is the sum of the lengths of the operands. However, the length of the result is two bytes less if redundant shift code characters are eliminated from the result. Redundant shift code characters exist when both character strings are EBCDIC mixed data, and the first string ends with a "shift-in" character (X'0F') and the second operand begins with a "shift-out" character (X'0E'). These two shift code characters are removed from the result.

The CCSID of the result is determined by the rules set forth in "Character conversion in set operations and concatenations" on page 816. Some consequences of those rules are the following:

- If either operand is BIT data, the result is BIT data.
- The conversion that occurs when SBCS data is compared with mixed data depends on the encoding scheme. If the encoding scheme is Unicode, the SBCS operand is converted to MIXED. Otherwise, the conversion depends on the field MIXED DATA on installation panel DSNTIPF for the DB2 that does the comparison:
  - Mixed data if the MIXED DATA option at the server is YES. The result is not necessarily well-formed mixed data.
  - SBCS data if the MIXED DATA option at the server is NO. If the mixed data cannot be converted to pure SBCS data, an error occurs.

If an operand is a string from a column with a field procedure, the operation applies to the decoded form of the value. The result does not inherit the field procedure.

One operand of concatenation can be a parameter marker. When one operand is a parameter marker, its data type and length attributes are considered to be the same as those for the operand that is not a parameter marker except for a string data

type. Refer to Table 43 on page 251 for the formula used to calculate data type length for untyped parameter markers in the CONCAT operator when another operand is a string data type. The order of concatenation operations must be considered to determine these attributes in the case of nested concatenation.

No operand of concatenation can be a distinct type even if the distinct type is based on a character data type. To concatenate a distinct type, create a user-defined function that is sourced on the CONCAT operator. For example, if distinct types TITLE and TITLE_DESCRIPTION were both sourced on data type VARCHAR(25), the following user-defined function, named ATTACH, could be used to concatenate the two distinct types:

```
CREATE FUNCTION ATTACH (TITLE, TITLE_DESCRIPTION)
   RETURNS VARCHAR(50) SOURCE CONCAT (VARCHAR(), VARCHAR())
```

Alternatively, the concatenation operator could be overloaded by using a user-defined function to add the distinct types:

```
CREATE FUNCTION "||" (TITLE, TITLE_DESCRIPTION)
   RETURNS VARCHAR(50) SOURCE CONCAT (VARCHAR(), VARCHAR())
```

Related concepts:

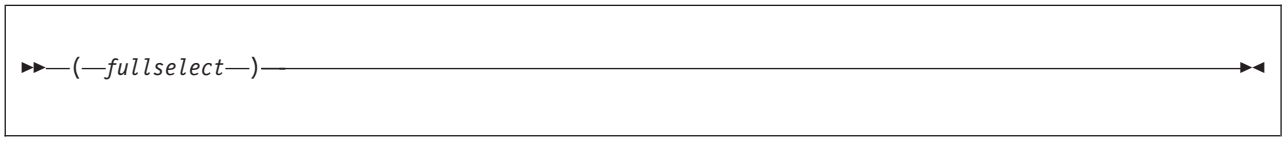
Concatenation of strings (Introduction to DB2 for z/OS)

Related reference:

"CONCAT" on page 419

Scalar-fullselect

A *scalar-fullselect* as supported in an expression is a fullselect, enclosed in parentheses, that returns a single row consisting of a single column value. If the fullselect does not return a row, the result of the expression is the null value. If more than one row is to be returned for a scalar fullselect, an error occurs.



If a set operator is not specified in the outermost fullselect and the select list element is an expression that is simply a column name, the result column name is based on the name of the column. Otherwise, the result column is unnamed.

If a column mask is defined to mask the column values in the final result, and if a column mask is applied to the column in the select list of a *scalar-fullselect*, the result of the *scalar-fullselect* must not be derived using set operator EXCEPT or INTERSECT. See Chapter 4, "Queries," on page 761 for more information about how column access controls affect a fullselect.

A scalar fullselect cannot be used in the following instances:

- A CHECK constraint in CREATE TABLE and ALTER TABLE statements
- A CREATE VIEW statement where the view definition includes the WITH CHECK option
- A CREATE FUNCTION (SQL) statement (subselect already restricted from the expression in the RETURN clause)
- An argument in a CALL statement for an input parameter

- An argument to an aggregate function, other than the *XML-expression* argument of the XMLAGG function
- An ORDER BY clause
- A GROUP BY clause
- A join-condition of the ON clause for INNER and OUTER JOINs

If the scalar fullselect is a subselect, it is also referred to as a scalar subselect. See "subselect" on page 764 for more information.

The following examples illustrate the use of *scalar-fullselect*. Assume that four tables (PARTS, PRODUCTS, PARTPRICE, and PARTINVENTORY) contain product data.

Example 1 - ***scalar-fullselect*** **in a WHERE clause:**
Find which products have the prices in the range of at least twice the lowest price of all the products and at most half the price of all the products.

```
SELECT PRODUCT, PRICE FROM PRODUCTS A
    WHERE
     PRICE BETWEEN 2 * (SELECT MIN(PRICE) FROM PRODUCTS)
           AND .5 * (SELECT MAX(PRICE) FROM PRODUCTS);
```

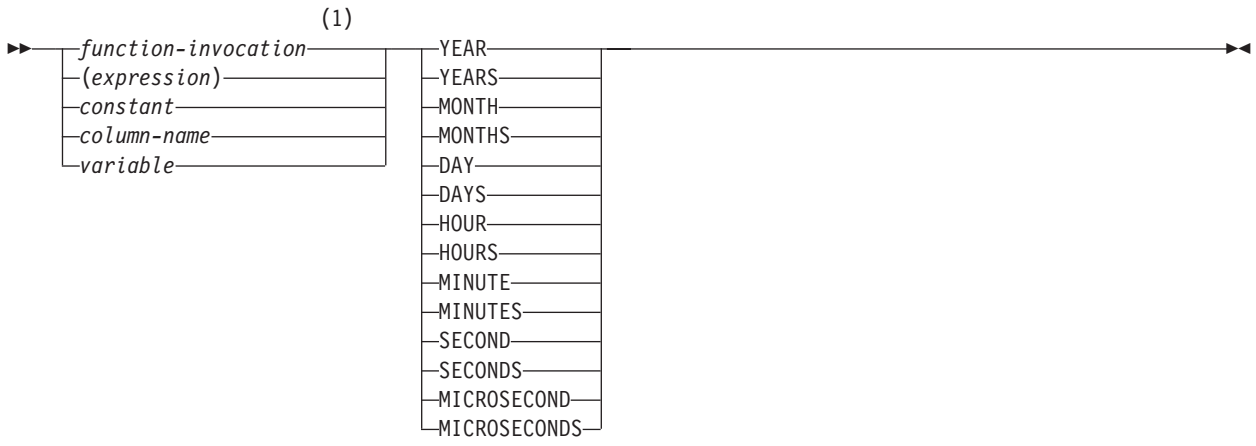
Example 2 - ***scalar-fullselect*** **in a SELECT list:**
For each part, find its price and its inventory.

```
SELECT PART,
      (SELECT PRICE FROM PARTPRICE WHERE PART=A.PART),
      (SELECT ONHAND# FROM INVENTORY WHERE PART=A.PART)
   FROM PARTS A;
```

Datetime operands and durations

Datetime values can be incremented, decremented, and subtracted. These operations can involve decimal numbers called durations. A *duration* is a positive or negative number representing an interval of time.

Labeled durations
The form a labeled duration is as follows:



Notes:

1 Must be a scalar function.

A *labeled duration* represents a specific unit of time as expressed by a number (which can be the result of an expression) followed by one of the seven duration keywords.[14] The number specified is converted as if it were assigned to a DECIMAL(15,0) number, except for SECONDS, which uses DECIMAL(27,12) to allow 0 to 12 fractional second digits to be included.

A labeled duration can only be used as an operand of an arithmetic operator in which the other operand is a value of the data type of date, time, or timestamp. Thus, the expression HIREDATE + 2 MONTHS + 14 DAYS is valid, whereas the expression HIREDATE + (2 MONTHS + 14 DAYS) is not. In both of these expressions, the labeled durations are 2 MONTHS and 14 DAYS.

Date duration

A *date duration* represents a number of years, months, and days expressed as a DECIMAL(8,0) number. To be properly interpreted, the number must have the format *yyyymmdd*, where *yyyy* represents the number of years, *mm* the number of months, and *dd* the number of days. The result of subtracting one DATE value from another, as in the expression HIREDATE - BIRTHDATE, is a date duration.

Time duration

A *time duration* represents a number of hours, minutes, and seconds expressed as a DECIMAL(6,0) number. To be properly interpreted, the number must have the format *hhmmss*, where *hh* represents the number of hours, *mm* the number of minutes, and *ss* the number of seconds. The result of subtracting one TIME value from another is a time duration.

Timestamp duration

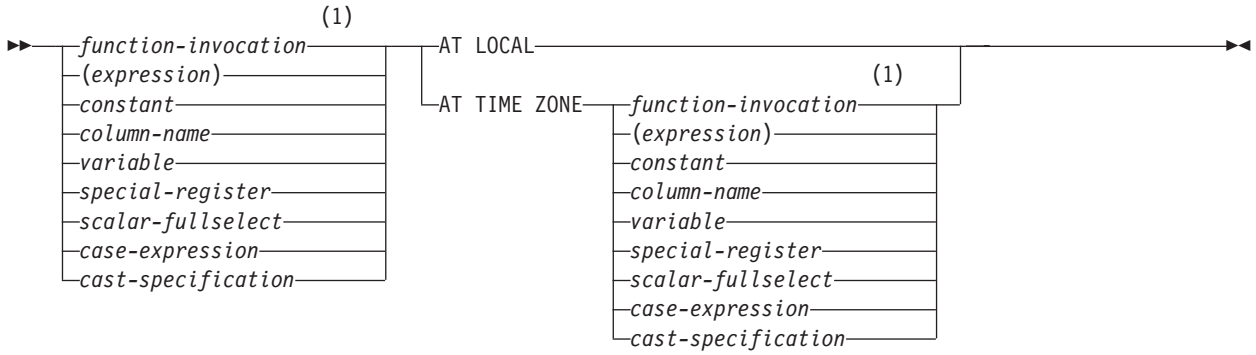
A *timestamp duration* represents a number of years, months, days, hours, minutes, seconds, and fractional seconds expressed as a DECIMAL(14+*s*,*s*) number, where s is the number of fractional seconds in the range from 0 to 12. To be interpreted properly, the number must have the format *yyyyxxddhhmmss*.*zzzzzzzzzzzz*, where *yyyy*, *xx*, *dd*, *hh*, *mm*, *ss*, and *zzzzzzzzzzzz* represent, respectively, the number of years, months, days, hours, minutes, seconds, and fractional seconds. The result of subtracting one timestamp value from another is a timestamp duration with a scale that matches the maximum timestamp precision of the timestamp operands.

Time zone specific expressions

Time zone specific expressions can be used to adjust timestamp values and character-string or graphic-string representations of timestamp values to specific time zones.

14. The singular form of these keywords is also acceptable: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, and MICROSECOND.

time-zone-specific-expressions



Notes:

1 Must be a scalar function.

The first operand for *time-zone-specific-expression* must be an expression that returns the value of either a built-in timestamp or a built-in character or graphic string data type. If the first operand is a character string or graphic string, it must not be a CLOB or DBCLOB value and its value must be a valid character-string or graphic-string representation of a timestamp. For the valid formats of string representations of datetime values, see "String representations of datetime values" on page 101.

If the first operand of *time-zone-specific-expression* returns a TIMESTAMP WITHOUT TIME ZONE value, the expression is implicitly cast to TIMESTAMP WITH TIME ZONE before being adjusted to the indicated time zone.

AT LOCAL
Specifies that the timestamp value is to be adjusted for the local time zone using the SESSION TIME ZONE special register.

AT TIME ZONE
Specifies that the timestamp is to be adjusted for the time zone that is represented by the expression.

expression is a character or graphic string. It must not be a CLOB or DBCLOB value, and its value must be left justified and be of the form '±*th*:*tm*', where *th* represents the time zone hour between -12 and +14, and *tm* represents the time zone minutes between 0 and 59, with values ranging from -12:59 to +14:00. The value must not be the null value.

The expression returns a TIMESTAMP WITH TIME ZONE value in the indicated time zone.

Syntax alternatives: TIMEZONE can be specified as an alternative to TIME ZONE.

Cast a timestamp for April 12, 2010 to the local time zone. Assume that the IMPLICIT TIME ZONE system parameter is set to '-8:00'.

```
CAST('2010-04-12-10:30:00.0 -5:00' AT LOCAL AS TIMESTAMP)
```

Returns: `2010-04-12-07:30:00.000000`.

Insert a timestamp value with a time zone into a table, tz, and retrieve it as a timestamp with the local time zone, with +08:00, and adjusted for UTC. Assume that table tz exists as follows:

```
CREATE TABLE tz(tstz TIMESTAMP WITH TIME ZONE);

INSERT INTO tz(tstz) VALUES(TIMESTAMP '2010-01-01-10.23.51-08:00');
```

1. Retrieve the value of the tstz column adjusted for the local time:

   ```
   SELECT tstz AT LOCAL
    FROM SYSIBM.SYSDUMMY1;
   ```

2. Retrieve the value of the tstz column adjusted for the time zone +08:00:

   ```
   SELECT tstz AT TIME ZONE '+08:00'
    FROM SYSIBM.SYSDUMMY1;
   ```

3. Retrieve the value of the tstz column adjusted for UTC:

   ```
   SELECT tstz AT TIME ZONE '00:00'
    FROM SYSIBM.SYSDUMMY1;
   ```

Datetime arithmetic in SQL

The only arithmetic operations that can be performed on datetime values are addition and subtraction.

If a datetime value is the operand of addition, the other operand must be a duration. The specific rules governing the use of the addition operator with datetime values follow.

- If one operand is a date, the other operand must be a date duration or labeled duration of years, months, or days.
- If one operand is a time, the other operand must be a time duration or a labeled duration of hours, minutes, or seconds.
- If one operand is a timestamp, the other operand must be a duration. Any type of duration is valid.
- Neither operand of the addition operator can be a parameter marker. For a discussion of parameter markers, see Parameter markers in “PREPARE” on page 1781.

The rules for the use of the subtraction operator on datetime values are not the same as those for addition because a datetime value cannot be subtracted from a duration, and because the operation of subtracting two datetime values is not the same as the operation of subtracting a duration from a datetime value. The specific rules governing the use of the subtraction operator with datetime values follow.

- If the first operand is a date, the second operand must be a date, a date duration, a string representation of a date, or a labeled duration of years, months, or days.
- If the second operand is a date, the first operand must be a date, or a string representation of a date.
- If the first operand is a time, the second operand must be a time, a time duration, a string representation of a time, or a labeled duration of hours, minutes, or seconds.
- If the second operand is a time, the first operand must be a time, or string representation of a time.
- If the first operand is a timestamp, the second operand must be a timestamp, a string representation of a timestamp, or a duration.
- If the second operand is a timestamp, the first operand must be a timestamp or a string representation of a timestamp.

- Neither operand of the subtraction operator can be a parameter marker.

When an operand in a datetime expression is a string, it might undergo character conversion before it is interpreted and converted to a datetime value. When its CCSID is not that of the default for mixed strings, a mixed string is converted to the default mixed data representation. When its CCSID is not that of the default for SBCS strings, an SBCS string is converted to the default SBCS representation.

Date arithmetic

Date values can be subtracted, incremented, or decremented.

Subtracting dates: The result of subtracting one date (DATE2) from another (DATE1) is a date duration that specifies the number of years, months, and days between the two dates. The data type of the result is DECIMAL(8,0). If DATE1 is greater than or equal to DATE2, DATE2 is subtracted from DATE1. If DATE1 is less than DATE2, however, DATE1 is subtracted from DATE2, and the sign of the result is made negative. The following procedural description clarifies the steps involved in the operation RESULT = DATE1 - DATE2.

Date subtraction: result = date1 - date2

- If `DAY(DATE2) <= DAY(DATE1)` then `DAY(RESULT) = DAY(DATE1) - DAY(DATE2)`
- If `DAY(DATE2) > DAY(DATE1)` then `DAY(RESULT) =` *N* `+ DAY(DATE1) - DAY(DATE2)` where *N* = the last day of `MONTH(DATE2)`. `MONTH(DATE2)` is then incremented by 1.
- If `MONTH(DATE2) <= MONTH(DATE1)` then `MONTH(RESULT) = MONTH(DATE1) - MONTH(DATE2)`
- If `MONTH(DATE2) > MONTH(DATE1)` then `MONTH(RESULT) = 12 + MONTH(DATE1) - MONTH(DATE2)` and `YEAR(DATE2)` is incremented by 1.
- `YEAR(RESULT) = YEAR(DATE1) - YEAR(DATE2)`

For example, the result of DATE('3/15/2005') - '12/31/2004' is 215 (or, a duration of 0 years, 2 months, and 15 days). In this example, notice that the second operand did not need to be converted to a date. According to one of the rules for subtraction, described under "Datetime arithmetic in SQL" on page 257, the second operand can be a string representation of a date if the first operand is a date.

Incrementing and decrementing dates: The result of adding a duration to a date, or of subtracting a duration from a date, is itself a date. (For the purposes of this operation, a month denotes the equivalent of a calendar page. Adding months to a date, then, is like turning the pages of a calendar, starting with the page on which the date appears.) The result must fall between the dates January 1, 0001 and December 31, 9999 inclusive. If a duration of years is added or subtracted, only the year portion of the date is affected. The month is unchanged, as is the day unless the result would be February 29 of a non-leap-year. Here the day portion of the result is set to 28, and the SQLWARN6 field of the SQLCA is set to W, indicating that an end-of-month adjustment was made to correct an invalid date. *DB2 Application Programming and SQL Guide* also describes how SQLWARN6 is set.

Similarly, if a duration of months is added or subtracted, only months and, if necessary, years are affected. The day portion of the date is unchanged unless the result would be invalid (September 31, for example). In this case the day is set to the last day of the month, and the SQLWARN6 field of the SQLCA is set to W to indicate the adjustment.

Adding or subtracting a duration of days will, of course, affect the day portion of the date, and potentially the month and year. Adding or subtracting a duration of days will not cause an end-of-the-month adjustment.

Date durations, whether positive or negative, can also be added to and subtracted from dates. As with labeled durations, the result is a valid date, and SQLWARN6 is set to W to indicate any necessary end-of-month adjustment.

When a positive date duration is added to a date, or a negative date duration is subtracted from a date, the date is incremented by the specified number of years, months, and days, in that order. Thus, DATE1+*X*, where *X* is a positive DECIMAL(8,0) number, is equivalent to the expression:

```
DATE1 + YEAR(X) YEARS + MONTH(X) MONTHS + DAY(X) DAYS
```

When a positive date duration is subtracted from a date, or a negative date duration is added to a date, the date is decremented by the specified number of days, months, and years, in that order. Thus, DATE1-*X*, where *X* is a positive DECIMAL(8,0) number, is equivalent to the expression:

```
DATE1 - DAY(X) DAYS - MONTH(X) MONTHS - YEAR(X) YEARS
```

Adding a month to a date gives the same day one month later unless that day does not exist in the later month. In that case, the day in the result is set to the last day of the later month. For example, January 28 plus one month gives February 28; one month added to January 29, 30, or 31 results in either February 28 or, for a leap year, February 29. If one or more months is added to a given date and then the same number of months is subtracted from the result, the final date is not necessarily the same as the original date.

If one or more months are added to a given date and then the same number of months is subtracted from the result, the final date is not necessarily the same as the original date. In addition, logically equivalent expressions might not produce the same result. For example, the following two expressions do not produce the same result:

```
(DATE('2005 01 31') + 1 MONTH) + 1 MONTH    -- results in 2005-03-28
DATE('2005 01 31') + 2 MONTHS               -- results in 2005-03-31
```

The order in which labeled date durations are added to and subtracted from dates can affect the results. When you add labeled date durations to a date, specify them in the order of YEARS + MONTHS + DAYS. When you subtract labeled date durations from a date, specify them in the order of DAYS - MONTHS - YEARS. For example, to add one year and one day to a date, specify:

```
DATE1 + 1 YEAR + 1 DAY
```

To subtract one year, one month, and one day from a date, specify:

```
DATE1 - 1 DAY - 1 MONTH - 1 YEAR
```

Time arithmetic

Times can be subtracted, incremented, or decremented.

Subtracting times: The result of subtracting one time (*TIME2*) from another (*TIME1*) is a time duration that specifies the number of hours, minutes, and seconds between the two times. The data type of the result is DECIMAL(6,0). If *TIME1* is greater than or equal to *TIME2*, *TIME2* is subtracted from *TIME1*. If *TIME1* is less than *TIME2*, however, *TIME1* is subtracted from *TIME2*, and the sign

of the result is made negative. The following procedural description clarifies the steps involved in the operation *RESULT* = *TIME1* - *TIME2*.

Time subtraction: ***result* = *time1* - *time2***

- If SECOND(*TIME2*) <= SECOND(*TIME1*) then SECOND(*RESULT*) = SECOND(*TIME1*) - SECOND(*TIME2*).
- If SECOND(*TIME2*) > SECOND(*TIME1*) then SECOND(*RESULT*) = 60 + SECOND(*TIME1*) - SECOND(*TIME2*) and MINUTE(*TIME2*) is incremented by 1.
- If MINUTE(*TIME2*) <= MINUTE(*TIME1*) then MINUTE(*RESULT*) = MINUTE(*TIME1*) - MINUTE(*TIME2*).
- If MINUTE(*TIME2*) > MINUTE(*TIME1*) then MINUTE(*RESULT*) = 60 + MINUTE(*TIME1*) - MINUTE(*TIME2*) and HOUR(*TIME2*) is incremented by 1.
- HOUR(*RESULT*) = HOUR(*TIME1*) - HOUR(*TIME2*).

For example, the result of TIME('11:02:26') - '00:32:56' is '102930' (a duration of 10 hours, 29 minutes, and 30 seconds). In this example, notice that the second operand did not need to be converted to a time. According to one of the rules for subtraction, described under "Datetime arithmetic in SQL" on page 257, the second operand can be a string representation of a time if the first operand is a time.

Incrementing and decrementing times: The result of adding a duration to a time, or of subtracting a duration from a time, is itself a time. Any overflow or underflow of hours is discarded, thereby ensuring that the result is always a time. If a duration of hours is added or subtracted, only the hours portion of the time is affected. Adding 24 hours to the time '00:00:00' results in the time '24:00:00'. However, adding 24 hours to any other time results in the same time; for example, adding 24 hours to the time '00:00:59' results in the time '00:00:59'. The minutes and seconds are unchanged.

Similarly, if a duration of minutes is added or subtracted, only minutes and, if necessary, hours are affected. The seconds portion of the time is unchanged.

Adding or subtracting a duration of seconds affects the seconds portion of the time and might affect the minutes and hours.

Time durations, whether positive or negative, can also be added to and subtracted from times. The result is a time that has been incremented or decremented by the specified number of hours, minutes, and seconds, in that order. Thus, *TIME1* + *X*, where *X* is a positive DECIMAL(6,0) number, is equivalent to the expression

```
TIME1 + HOUR(X) HOURS + MINUTE(X) MINUTES + SECOND(X) SECONDS
```

Timestamp arithmetic

Timestamps can be subtracted, incremented, or decremented.

If any of the operands are TIMESTAMP WITH TIME ZONE, any TIMESTAMP WITHOUT TIME ZONE values are implicitly cast to TIMESTAMP WITH TIME ZONE, and the datetime arithmetic operation is performed in UTC time (ignoring the time zone).

Subtracting timestamps: The result of subtracting one timestamp (*TS2*) from another (*TS1*) is a timestamp duration that specifies the number of years, months, days, hours, minutes, seconds, and fractional seconds between the two timestamps.

The data type of the result is DECIMAL(14+*s*,*s*), where *s* is the maximum timestamp precision of *TS1* and *TS2*. If *TS1* is greater than or equal to *TS2*, *TS2* is

subtracted from *TS1*. If *TS1* is less than *TS2*. However, *TS1* is subtracted from *TS2* and the sign of the result is made negative. A subtraction that involves a timestamp with a time zone operand is based on the UTC value of the timestamp with the time zone. The time zone is ignored.

The following procedural description clarifies the steps involved in the operation *RESULT = TS1 - TS2*.

Timestamp subtraction: *result = ts1 - ts2*

- If MICROSECOND(*TS2*) <= MICROSECOND(*TS1*) then MICROSECOND(*RESULT*) = MICROSECOND(*TS1*) - MICROSECOND(*TS2*).
- If MICROSECOND(*TS2*) > MICROSECOND(*TS1*) then MICROSECOND(*RESULT*) = 1000000 + MICROSECOND(*TS1*)- MICROSECOND(*TS2*) and SECOND(*TS2*) is incremented by 1.
- If SECOND(*TS2*,*s*) <= SECOND(*TS1*,*s*) then SECOND(*RESULT*,*s*) = SECOND(*TS1*,*s*) - SECOND(*TS2*,*s*).
- If SECOND(*TS2*,*s*) > SECOND(*TS1*,*s*) then SECOND(*RESULT*,*s*) = 60 + SECOND(*TS1*,*s*) – SECOND(*TS2*,*s*).

  MINUTE(*TS2*) is incremented by 1.
- If HOUR(*TS2*) <= HOUR(*TS1*) then HOUR(*RESULT*) = HOUR(*TS1*) - HOUR(*TS2*).
- If HOUR(*TS2*) > HOUR(*TS1*) then HOUR(*RESULT*) = 24 + HOUR(*TS1*) - HOUR(*TS2*) and DAY(*TS2*) is incremented by 1.

The minutes part of the timestamps are subtracted as specified in the rules for subtracting times.

The date part of the timestamps is subtracted as specified in the rules for subtracting dates.

Incrementing and decrementing timestamps: The result of adding a duration to a timestamp, or of subtracting a duration from a timestamp, is itself a timestamp. The precision of the result timestamp matches the precision of the timestamp operand. The date and time arithmetic is performed as previously defined, except that an overflow or underflow of hours is carried into the date part of the result, which must be within the range of valid dates. The time arithmetic portion is similar to time arithmetic, except that it also considers the fractional seconds included in the duration. For example, subtracting a duration, *X*, from a timestamp, TIMESTAMP1, where *X* is a DECIMAL(14+s,s) number, is equivalent to the expression:

```
TIMESTAMP1 - YEAR(X)  YEARS - MONTH(X)  MONTHS - DAY(X)  DAYS
           - HOUR(X)  HOURS - MINUTE(X)  MINUTES - SECOND(X, s)  SECONDS
```

When subtracting a duration with a non-zero scale or a labeled duration of SECOND or SECONDS with a value that includes fractions of a second, the subtraction is performed as if the timestamp value has up to 12 fractional second digits. The resulting value is assigned to a timestamp value with the timestamp precision of the timestamp operand, which could result in truncation of fractional second digits.

When the result of an operation is midnight, the time portion of the result can be '24.00.00' or '00.00.00'. A comparison of those two values does not result in 'equal'. Microseconds overflow into seconds.

Precedence of operations

Expressions within parentheses are evaluated first. When the order of evaluation is not specified by parentheses, prefix operators are applied before multiplication and

division, and multiplication, division, and concatenation are applied before addition and subtraction. Operators at the same precedence level are applied from left to right.

Example 1: In this example, the first operation is the addition in (SALARY + BONUS) because it is within parenthesis. The second operation is multiplication because it is a higher precedence level than the second addition operator and it is to the left of the division operator. The third operation is division because it is at a higher precedence level than the second addition operator. Finally, the remaining addition is performed.

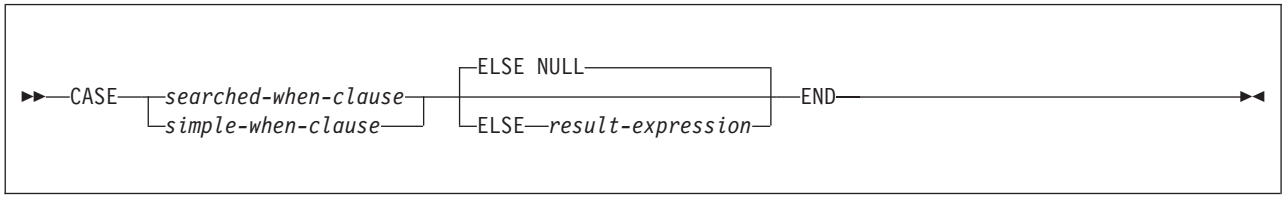
```
1.10 * (SALARY + BONUS) + SALARY / :VAR3
    (2)       (1)      (4)      (3)
```

Example 2: In this example, the first operation (CONCAT) combines the character strings in the variables *YYYYMM* and *DD* into a string representing a date. The second operation (-) then subtracts that date from the date being processed in DATECOL. The result is a date duration that indicates the time elapsed between the two dates.

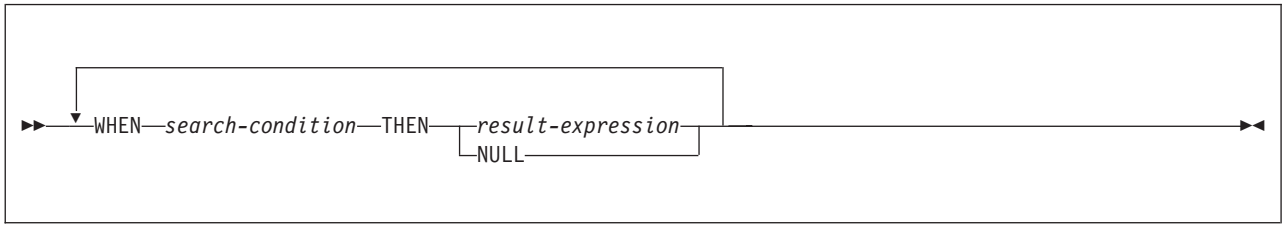
```
DATECOL - :YYYYMM CONCAT :DD
       (2)         (1)
```

CASE expressions

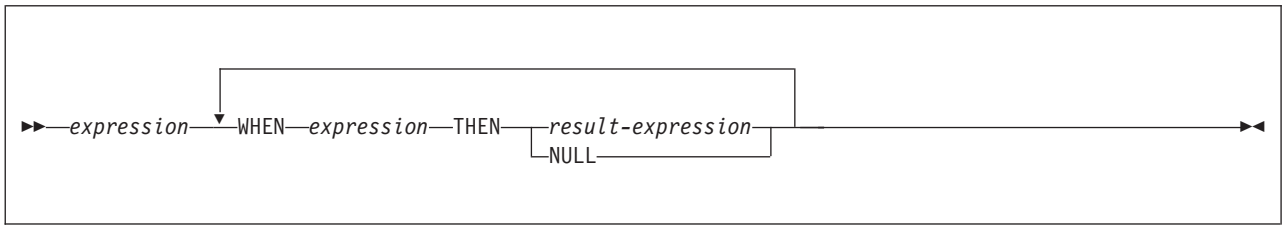
A CASE expression allows an expression to be selected based on the evaluation of one or more conditions.



searched-when-clause:



simple-when-clause:



In general, the value of the *case-expression* is the value of the *result-expression* following the first (leftmost) *when-clause* that evaluates to true. If no case evaluates to true and the ELSE keyword is present, the result is the value of the *result-expression* or NULL. If no case evaluates to true and the ELSE keyword is not present, the result is NULL. When a case evaluates to unknown (because of NULL values), the case is NOT true and hence is treated the same way as a case that evaluates to false.

searched-when-clause
Specifies a *search-condition* that is applied to each row or group of table data presented for evaluation, and the result when that condition is true.

Pair-wise comparison is performed. Implicit cast of each pair follows the same rule as for a basic predicate. The *searched-when-clause* performs implicit cast on string and numeric search conditions.

simple-when-clause
Specifies that the value of the *expression* prior to the first WHEN keyword is tested for equality with the value of each *expression* that follows the WHEN keyword. It also specifies the result for when that condition is true.

The data type of the *expression* prior to the first WHEN keyword must be compatible with the data types of the *expression* that follows each WHEN

keyword. The data type of any of the expressions cannot be a CLOB, DBCLOB or BLOB. In addition, the *expression* prior to the first WHEN keyword cannot include a function that is not deterministic or has an external action.

result-expression **or NULL**
Specifies the value that follows the THEN and ELSE keywords. It specifies the result of a *searched-when-clause* or a *simple-when-clause* that is true, or the result if no case is true. There must be at least one *result-expression* in the CASE expression with a defined data type. NULL cannot be specified for every case.

All *result-expression*s must have compatible data types. The attributes of the result are determined according to the rules that are described in "Rules for result data types" on page 144. When the result is a string, its attributes include a CCSID. For the rules on how the CCSID is determined, see "Determining the encoding scheme and CCSID of a string" on page 47.

search-condition
Specifies a condition that is true, false, or unknown about a row or group of table data. The *search-condition* can be a predicate, including predicates that contain fullselects (scalar or non-scalar) or row-value expressions.

If *search-condition* in a *searched-when-clause* specifies a quantified predicate or an IN predicate that includes a fullselect, the CASE expression cannot be used in the following contexts:

- select lists
- a VALUES clause of an INSERT or MERGE statement
- a SET or assignment clause of an UPDATE, MERGE, or DELETE statement
- the right side of a SET or assignment statement
- the definition of a column mask or a row permission

If *search-condition* in a *searched-when-clause* specifies an EXISTS predicate, the CASE expression cannot be used in the following contexts:

- a VALUES clause of an INSERT or MERGE statement
- the right side of a SET or assignment statement

END
Ends a *case-expression*.

If a CASE expression is in a select list that derives the final result table, and if the *simple-when-clause* or the *searched-when-clause* references a basic predicate with a fullselect, column masks cannot be applied to the columns in the THEN clauses which derive the result of the CASE expression.

If a CASE expression is in a select list that derives the final result table, and if the *simple-when-clause* or *searched-when-clause* references a column for which column access control is activated, the column mask cannot be applied to the column and an error is returned.

If a CASE expression is in a SET clause of an UPDATE, MERGE, or DELETE statement, a VALUES clause of an INSERT or MERGE statement, or the fullselect of an INSERT from a fullselect, and if the *simple-when-clause* or the *searched-when-clause* references a column for which column access control is activated, the column access control is ignored for the column.

Two scalar functions, NULLIF and COALESCE, are specialized to handle a subset of the functionality provided by CASE. The following table shows the equivalent expressions using CASE or these functions.

Table 44. Equivalent case expressions

CASE expression	Equivalent expression
CASE WHEN e1=e2 THEN NULL ELSE e1 END	NULLIF(e1,e2)
CASE WHEN e1 IS NOT NULL THEN e1 ELSE e2 END	COALESCE(e1,e2)
CASE WHEN e1 IS NOT NULL THEN e1 ELSE COALESCE(e2,...,eN) END	COALESCE(e1,e2,...,eN)

Example 1 (simple-when-clause): Assume that in the EMPLOYEE table the first character of a department number represents the division in the organization. Use a CASE expression to list the full name of the division to which each employee belongs.

```
SELECT EMPNO, LASTNAME,
    CASE SUBSTR(WORKDEPT,1,1)
    WHEN 'A' THEN 'Administration'
    WHEN 'B' THEN 'Human Resources'
    WHEN 'C' THEN 'Design'
    WHEN 'D' THEN 'Operations'
    END
FROM EMPLOYEE;
```

Example 2 (searched-when-clause): You can also use a CASE expression to avoid "division by zero" errors. From the EMPLOYEE table, find all employees who earn more than 25 percent of their income from commission, but who are not fully paid on commission:

```
SELECT EMPNO, WORKDEPT, SALARY+COMM FROM EMPLOYEE
WHERE (CASE WHEN SALARY=0 THEN 0
            ELSE COMM/(SALARY+COMM)
            END) > 0.25;
```

Example 3 (searched-when-clause): You can use a CASE expression to avoid "division by zero" errors in another way. The following queries show an accumulation or summing operation. In the first query, DB2 performs the division before performing the CASE statement and an error occurs along with the results.

```
SELECT REF_ID,PAYMT_PAST_DUE_CT,
    CASE
    WHEN PAYMT_PAST_DUE_CT=0 THEN 0
    WHEN PAYMT_PAST_DUE_CT>0 THEN
          SUM(BAL_AMT/PAYMT_PAST_DUE_CT)
         END
FROM PAY_TABLE
GROUP BY REF_ID,PAYMT_PAST_DUE_CT;
```

However, if the CASE expression is included in the SUM aggregate function, the CASE expression would prevent the errors. In the following query, the CASE expression screens out the unwanted division because the CASE operation is performed before the division.

```
SELECT REF_ID,PAYMT_PAST_DUE_CT,
    SUM(CASE
    WHEN PAYMT_PAST_DUE_CT=0 THEN 0
    WHEN PAYMT_PAST_DUE_CT>0 THEN
```

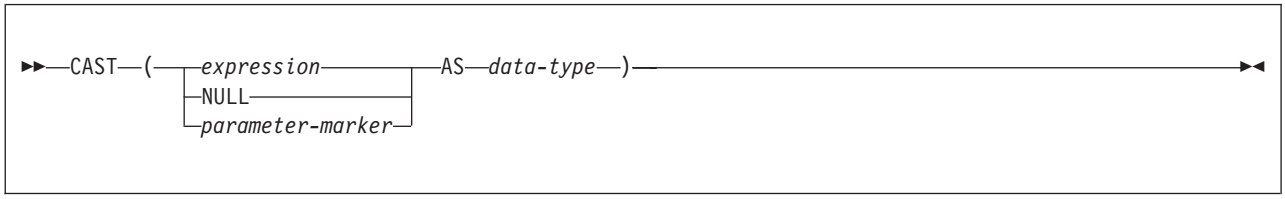
```
        BAL_AMT/PAYMT_PAST_DUE_CT
          END)
   FROM PAY_TABLE
 GROUP BY REF_ID,PAYMT_PAST_DUE_CT;
```

Example 4: This example shows how to group the results of a query by a CASE expression without having to re-type the expression. Using the sample employee table, find the maximum, minimum, and average salary. Instead of finding these values for each department, assume that you want to combine some departments into the same group.

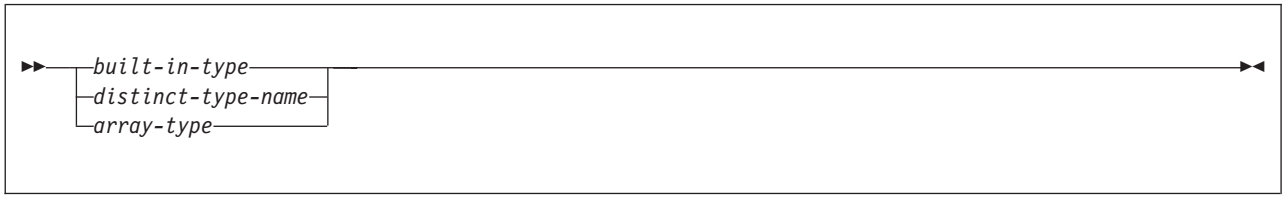
```
SELECT CASE_DEPT,MAX(SALARY),MIN(SALARY),AVG(SALARY)
FROM (SELECT SALARY,CASE WHEN WORKDEPT = 'A00' OR WORKDEPT = 'E21'
                           THEN 'A00_E21'
                         WHEN WORKDEPT = 'D11' OR WORKDEPT = 'E11'
                           THEN 'D11_E11'
                         ELSE WORKDEPT
                    END AS CASE_DEPT
      FROM DSN8B10.EMP) X
      GROUP BY CASE_DEPT;
```

CAST specification

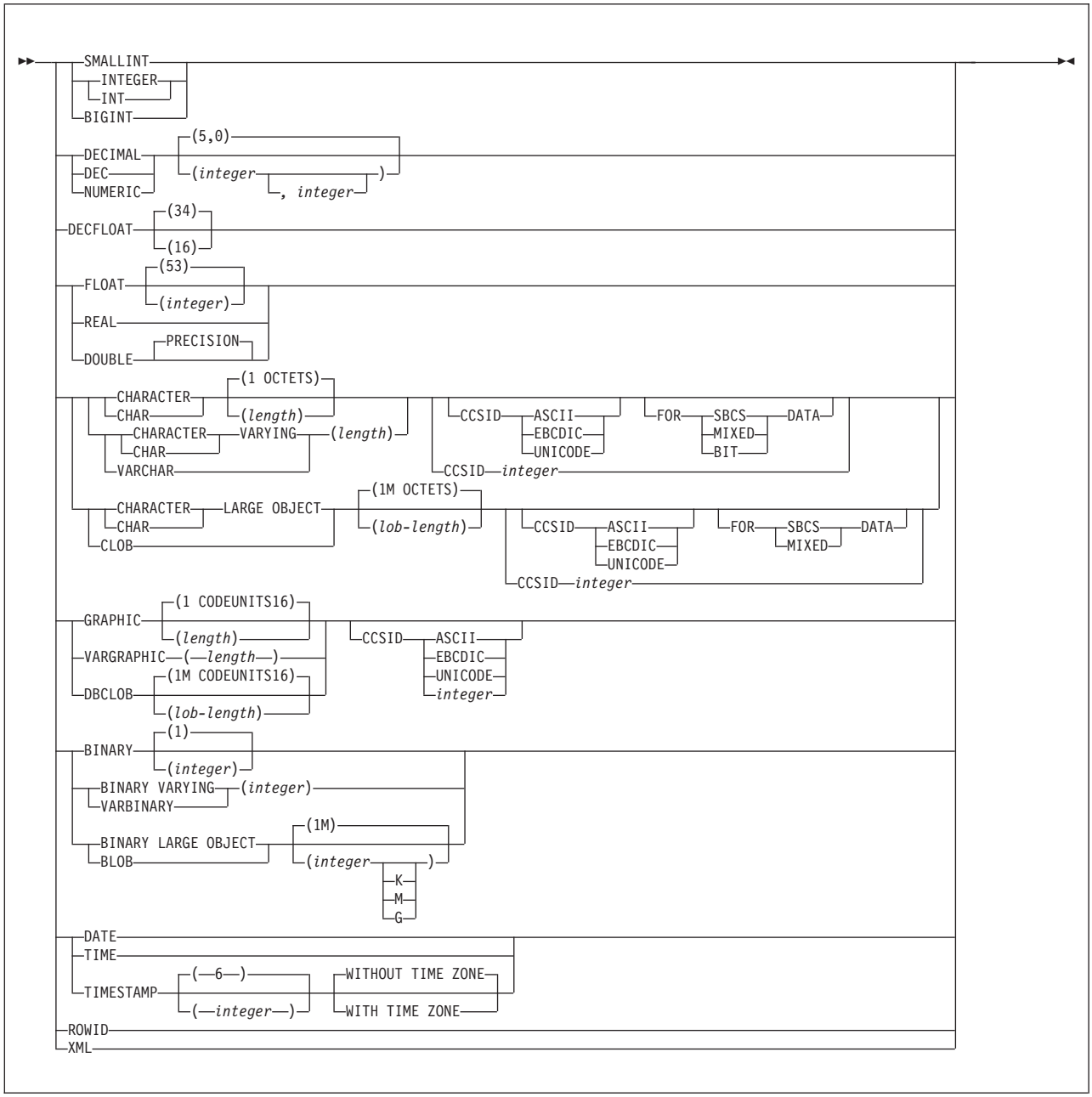
The CAST specification returns the first operand (the cast operand) converted to the data type that is specified by *data-type*.



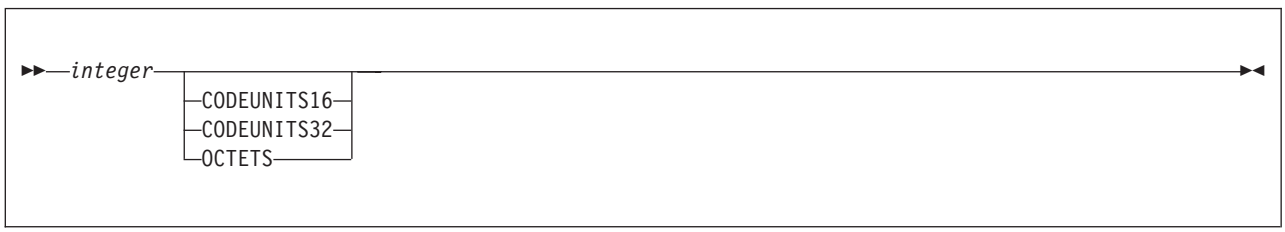
data-type:



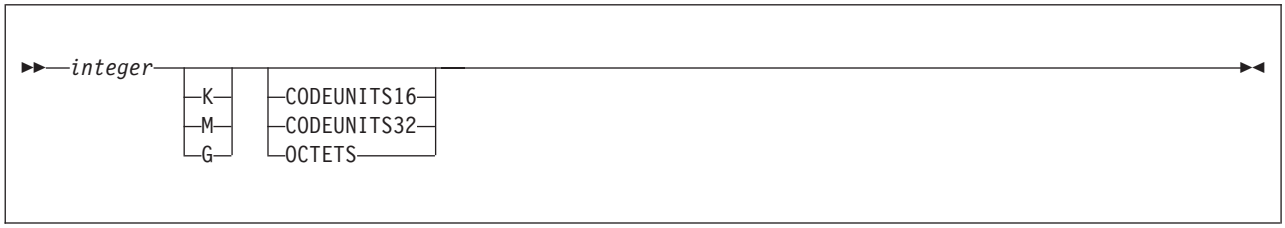
built-in-type:



length:



lob-length:



If the data type of either operand is a distinct type, the privilege set must implicitly include EXECUTE authority on the generated cast functions for the distinct type. The CAST specification allows the second operand to be cast to a particular encoding scheme or CCSID if the second operand represents character data. The CCSID clause can be specified following CHAR, VARCHAR, CLOB, GRAPHIC, VARGRAPHIC, and DBCLOB data types.

expression
Specifies that the cast operand is an expression other than NULL or a parameter marker. The result is the value of the operand value converted to the specified target *data type*.

The supported casts are shown in "Casting between data types" on page 111. If the cast is not supported, an error is returned.

When a character string is cast to a character string with a different length or a graphic string is cast to a graphic string with a different length, a warning occurs if any characters except trailing blanks are truncated. The warning also occurs if any characters are truncated when a BLOB operand is cast, or if the time zone characters are truncated when a TIMESTAMP WITH TIME ZONE operand is cast to a string

NULL
Specifies that the cast operand is null. The result is a null value with the specified target *data type*.

parameter-marker
A parameter marker, which is normally considered an expression, has a special meaning as a cast operand. When the cast operand is a *parameter-marker*, the *data type* that is specified represents the "promise" that the replacement value for the parameter marker will be assignable to the specified data type (using "store assignment" rules). Such a parameter marker is considered a *typed parameter marker*. Typed parameter markers are treated like any other typed value for the purpose of function resolution, a DESCRIBE of a select list, or column assignment.

data-type
Specifies the data type of the result. If the data type is not qualified, the SQL path is used to find the appropriate data type. For more information, see "SQL path" on page 64. For a description of *data-type*, see "CREATE TABLE" on page 1388. (For portability across operating systems, when specifying a floating-point data type, use REAL or DOUBLE instead of FLOAT.)

- If the cast operand is *expression*, see "Casting between data types" on page 111 and use any of the target data types that are supported for the data type of the cast operand.
- If the cast operand is NULL, you can use any data type.
- If the cast operand is a *parameter-marker*:

– If the target data type is a distinct type, the application that uses the parameter marker uses the source data type of the distinct type.
– If the target data type is an array type:
  - The elements in the source array value must be castable to the data type of the elements of the target array type. The index values for the source array value must be castable to the data type of the index of the target array type.
  - If the target array type is an ordinary array, the cardinality of the source array value must be less than or equal to the maximum cardinality of the target array type.
– Otherwise, any data type if valid.

array-type can only be specified as a target data type for a CAST specification that is within SQL PL.

length
Specifies the length of the result.

You can specify that the length of the result be evaluated in a specific number of string units: CODEUNITS16, CODEUNITS32, or OCTETS. If *expression* is a character string that is defined as bit data, CODEUNITS16, or CODEUNITS32 cannot be specified. If *expression* is a graphic string, OCTETS cannot be specified.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String unit specifications" on page 87.

lob-length
Specifies the length of the result.

You can specify that the length of the result be evaluated in a specific number of string units: CODEUNITS16, CODEUNITS32, or OCTETS. If *expression* is a graphic string, OCTETS cannot be specified.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String unit specifications" on page 87.

CCSID *encoding-scheme*
Specifies the encoding scheme for the target data type. The specific CCSIDs for SBCS, BIT, and MIXED data are determined by the default CCSIDs for the server for the specified encoding scheme. The valid values are ASCII, EBCDIC, and UNICODE.

CCSID *integer*
Specifies that the target data type be encoded using the CCSID *integer*. The value must be one of the CCSID values in DECP. If the second operand is CHAR, VARCHAR, or CLOB, the CCSID specified must be either a SBCS, or MIXED CCSID, or 65535 for bit data. If the second operand is GRAPHIC, VARGRAPHIC, or DBCLOB, the CCSID specified must be a DBCS CCSID. See Determining the CCSID of the result if neither CCSID *integer* nor CCSID *encoding-scheme* is specified. See Determining the CCSID of the result for special considerations regarding CCSID 367.

Interaction between length and CCSID clauses: If both the *length* and CCSID clauses are specified, the data is first cast to the specified CCSID, and then the *length* is applied. If either CODEUNITS16 or CODEUNITS32 is specified, the specification of length applies to the units specified. That is, the data is converted to an intermediate form (in Unicode), the length is applied, and the data is converted to the specified CCSID.

Resolution of cast functions: DB2 uses the implicit or explicit schema name and the data type name of *data-type*, and function resolution to determine the specific function to use to convert *expression* to *data-type*. See Qualified function resolution for more information.

Result of the CAST: When numeric data is cast to character data, the data type of the result is a fixed-length character string, which is similar to the result that the CHAR function would give. (For more information, see "CHAR" on page 398.) When character data is cast to numeric data, the data type of the result depends on the data type of the specified number. For example, character data that is cast to an integer becomes a large integer, which is similar to the result that the INTEGER function would give. (For more information see "INTEGER or INT" on page 496.)

If the data type of the result is character, the subtype of the result is determined as follows:

- If *expression* is graphic, the subtype of the result is mixed.
- If *expression* is a datetime data type, the subtype of the result is mixed. The exception is when the default encoding scheme is EBCDIC and there is no mixed or graphic data on the system for EBCDIC.
- If *expression* is a row ID and *data-type* is not CLOB, the result is bit data.
- If *expression* is character, the subtype of the result is the same as *expression*.
- Otherwise, the subtype depends on the encoding scheme of the result. If the encoding scheme of the result is not Unicode and the field MIXED DATA on installation panel DSNTIPF is NO, the subtype of the result is SBCS. Otherwise, the subtype of the result is mixed.

Casting constant values to DECFLOAT: To cast a constant value, where the value is negative zero, or a floating point constant to DECFLOAT, specify the value as a character string constant rather than a numeric constant. For example:

```
DECFLOAT('-0')              -- causes DB2 to retain the negative sign for a
                            --   value of negative zero
DECFLOAT('1.00E20')         -- causes DB2 to preserve the precision of the
                            --   floating point constant
```

Determining the CCSID and encoding scheme of the result: The CCSID of the result depends on whether the CCSID clause was specified and the context in which the CAST specification was specified.

If the CCSID clause was specified, the CCSID clause is used to determine the CCSID of the result as follows:

- If the CCSID clause was specified with EBCDIC, ASCII, or UNICODE, the clause determines the encoding scheme of the result. The CCSID of the result is the appropriate CCSID (from DECP) for that encoding scheme for the data type of the result.
- If the CCSID clause was specified with a numeric value representing bit data (65535), the CCSID of the result depends on the data type of the source. If the source data is not string data, the CCSID of the result is the appropriate CCSID for the application encoding scheme. See Note 1 in Table 4 on page 48. If the source is string data, the encoding scheme of the result is the same as the encoding scheme of expression, but the result is considered bit data.
- If the CCSID clause was specified with a numeric value, that number is the CCSID of the result. The encoding scheme of the result is determined from the numeric CCSID. In a CAST specification, CCSID 367 refers to ASCII data. For

example, assume that MYDATA is string data to be cast to CHAR(10). The following CAST specification returns ASCII SBCS data:

```
CAST(MYDATA AS CHAR(10) CCSID 367)
```

To explicitly cast the data to Unicode SBCS, use the following syntax:

```
CAST(MYDATA AS CHAR(10) CCSID UNICODE
      FOR SBCS DATA)
```

If the CCSID clause was not specified, the CCSID of the result is 65535 if the result is bit data. Otherwise, if the data type of the result is a character or graphic string data type, the encoding scheme and CCSID of the result are is determined as follows:

- If the *expression* and *data-type* are both character, the encoding scheme of the result is the same as *expression*. For example, assume CHAR_COL is a character column in the following sample:

  ```
  CAST(CHAR_COL AS VARCHAR(25))
  ```

  The result of the CAST is a varying length string with the same encoding scheme as the input. The CCSID of the result is the appropriate CCSID for the encoding scheme and subtype of the result.
- If the *expression* and *data-type* are both graphic, the encoding scheme and CCSID of the result is the same as *expression*.
- If the result is string and the *expression* is datetime, the result CCSID is the appropriate CCSID of the *expression* encoding scheme and the result subtype is the appropriate subtype of the CCSID.
- If the result is character, the encoding scheme and CCSID of the result depends on the context in which the CAST specification is specified:
  - If the statement follows the rules that are described for type 1 statements in "Determining the encoding scheme and CCSID of a string" on page 47, the CCSID is determined as follows:
    - If the statement references a table or view, the encoding scheme of that table or view determines the encoding scheme for the result.
    - Otherwise, the default EBCDIC encoding scheme is used for the result.

    The CCSID of the result is the appropriate CCSID for the encoding scheme and subtype of the result.
  - Otherwise, the CCSID of the result is the appropriate CCSID for the application encoding scheme and subtype of the result.
- If the result is graphic, the encoding scheme and the CCSID of the result depends on the context in which the CAST specification is specified:
  - If the statement follows the rules that are described for type 1 statements in "Determining the encoding scheme and CCSID of a string" on page 47, the CCSID is determined as follows:
    - If the statement references a table or view, the encoding scheme of that table or view determines the encoding scheme for the result.
    - Otherwise, the default EBCDIC encoding scheme is used for the result.

    The CCSID of the result is the appropriate CCSID for the encoding scheme and data type of the result.
  - Otherwise, the CCSID of the result is the appropriate CCSID for the application encoding scheme of the result.
- Otherwise, the CCSID of the result depends on the context in which the CAST specification was specified.

– If the statement follows the rules that are described for type 1 in statements in "Determining the encoding scheme and CCSID of a string" on page 47, the CCSID is determined as follows:
  - If the statement references a table or view, the encoding scheme of that table or view determines the encoding scheme for the result.
  - Otherwise, the default EBCDIC encoding scheme is used for the result.

  The CCSID of the result is the appropriate CCSID for the encoding scheme and data type of the result.

Alternative syntax for casting distinct types: There is alternative syntax for casting a distinct type to its source data type and vice versa. Assume that a distinct type D_MONEY was defined with the following statement and column MONEY was defined with that data type.

```
CREATE TYPE D_MONEY AS DECIMAL(9,2);
```

DECIMAL(MONEY) is equivalent syntax to CAST(MONEY AS DECIMAL(9,2)). Both forms of the syntax use the cast function that DB2 generated when the distinct type D_MONEY was created to convert the distinct type to its source type of DECIMAL(9,2).

However, it is possible that different cast functions might be chosen for the equivalent syntax forms because of the difference in function resolution, particularly the treatment on unqualified names. Although the process of function resolution is similar for both, in the CAST specification as described above, DB2 uses the schema name of the target data type to locate the function. Therefore, if an unqualified data type name is specified as the target data type, DB2 uses the SQL path to resolve the schema name of the distinct type and then searches for the function in that schema. In function notation, when an unqualified function name is specified, DB2 searches the schemas in the SQL path to find an appropriate function match, as described under "Function resolution" on page 234. For example, assume that you defined the following distinct types, which implicitly gives you both USAGE authority on the distinct types and EXECUTE authority on the cast functions that are generated for them:

```
CREATE TYPE SCHEMA1.AGE AS DECIMAL(2,0);
   one of the generated cast functions is:
   FUNCTION SCHEMA1.AGE(SYSIBM.DECIMAL(2,0)) RETURNS SCHEMA1.AGE
CREATE TYPE SCHEMA2.AGE AS INTEGER;
   one of the generated cast functions is:
   FUNCTION SCHEMA2.AGE(SYSIBM.INTEGER) RETURNS SCHEMA2.AGE
```

If *STU_AGE*, an INTEGER host variable, is cast to the distinct type with either of the following statements and the SQL path is SYSIBM, SCHEMA1, SCHEMA2:

```
Syntax 1:  CAST(:STU_AGE AS AGE);
Syntax 2:  AGE(:STU_AGE);
```

different cast functions are chosen. For syntax 1, DB2 first resolves the schema name of distinct type AGE as SCHEMA1 (the first schema in the path that contains a distinct type named AGE for which you have EXECUTE authority for the appropriate generated cast function). Then it looks for a suitable function in that schema and chooses SCHEMA1.AGE because the data type of *STU_AGE*, which is INTEGER, is promotable to the data type of the function argument, which is DECIMAL(2,0). For syntax 2, DB2 searches all the schemas in the path for an appropriate function and chooses SCHEMA2.AGE. DB2 selects SCHEMA2.AGE over SCHEMA1.AGE because the data type of its argument (INTEGER) is an exact

match for *STU_AGE* (INTEGER) and, therefore, a better match than the argument for SCHEMA1.AGE, which is DECIMAL(2,0).

Syntax alternatives: TIMEZONE can be specified as an alternative to TIME ZONE.

Example 1: Assume that an application needs only the integer portion of the SALARY column, which is defined as DECIMAL(9,2) from the EMPLOYEE table. The following query for the employee number and the integer value of SALARY could be prepared.

```
SELECT EMPNO, CAST(SALARY AS INTEGER) FROM EMPLOYEE;
```

Example 2: Assume that two distinct types exist in schema SCHEMAX. Distinct type D_AGE was based on SMALLINT and is the data type for the AGE column in the PERSONNEL table. Distinct type D_YEAR was based on INTEGER and is the data type for the RETIRE_YEAR column in the same table. The following UPDATE statement could be prepared.

```
UPDATE PERSONNEL SET RETIRE_YEAR =?
                 WHERE AGE = CAST( ? AS SCHEMAX.D_AGE);
```

The first parameter is an untyped parameter marker that has a data type of RETIRE_YEAR. However, the application will use an integer for the parameter marker. The parameter marker does not need to be cast because the SET is an assignment.

The second parameter marker is a typed parameter marker that is cast to the distinct type D_AGE. Casting the parameter marker satisfies the requirement that comparisons must be performed with compatible data types. The application will use the source data type, SMALLINT, to process the parameter marker.

Example 3: A CAST specification can be used to explicitly specify the data type of a parameter in a context where a parameter marker must be typed. In the following example, the CAST specification is used to tell DB2 to assume that the value that will be provided as input to the TIME function will be CHAR(20). See “PREPARE” on page 1781 for a list of contexts when invoking functions where parameter markers can be untyped. For all other contexts when invoking a function, the CAST specification can be used to explicitly specify the type of a parameter marker.

```
INSERT INTO ADMF001.CASTSQLJ VALUES( TIME(CAST(? AS CHAR(20)) ) )
```

Example 4: Assume that an application wants to cast an EBCDIC string to Unicode UTF-8. The string contains the value 'Jürgen', which is 6 bytes in ASCII or EBCDIC and is 7 bytes in Unicode UTF-8. In the following query, the CAST specification is invoked with the *length* clause with CODEUNITS32 specified to ensure that the data is not truncated. (In this case, CODEUNITS16 could also be specified as the string unit.)

```
SELECT CAST('Jürgen' AS VARCHAR(6 CODEUNITS32) CCSID UNICODE)
  FROM SYSIBM.SYSDUMMY1;
```

For this query, the data is converted from EBCDIC to Unicode UTF-16, the length clause is applied, and then the UTF-16 result is converted to UTF-8.

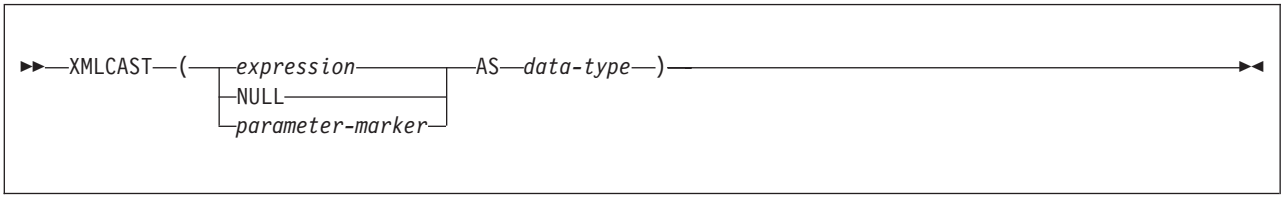
Example 5: When a keyword is used for a special value that is expressed as a constant in a context where the keyword could be interpreted as a name, the CAST specification can be used to explicitly cast the special value to decimal-floating point. Assume that MYTAB contains columns named C1 and INFINITY, and that you want to reference the decimal float-point value for infinity in the same SQL

statement. Use the CAST specification to explicitly cast INFINITY as a decimal floating-point value to ensure that it is not interpreted as the name of a column, parameter or variable:

```
SELECT INFINITY                                -- column named INFINITY
 FROM MYTAB
    WHERE C1 = CAST ('INFINITY' AS DECFLOAT) -- comparison is made with the
                                               -- decimal floating-point
                                               --  infinity value
```

XMLCAST specification

The XMLCAST specification returns the first operand (the cast operand) converted to the type specified by *data-type*.



XMLCAST supports casts involving XML values, including conversions between non-XML data types and the XML data type. Either the type of the cast operand or the specified data type must be XML. If both the type of the cast operand and the target data type are XML, XMLCAST acts as a no-op.

expression
If the cast operand is an expression, the result is the argument value converted to the specified target data type. The expression or the target data type must be the XML data type. *expression* cannot be a host variable or parameter marker.

NULL
If the cast operand is the NULL keyword, the target data type must be the XML data type. The result is a null XML value.

parameter-marker
If the cast operand is a parameter marker, the target data type must be the XML data type. A parameter marker (specified as a question mark character) is normally considered to be an expression, but in this case because it has special meaning. When the cast operand is a *parameter-marker*, the data type that is specified represents the "promise" that the replacement value for the parameter marker will be assignable to the specified data type (using assignment rules). Such a parameter marker is considered to be a typed parameter marker, which is treated like any other typed value for the purpose of function resolution, a describe operation on a select list, or column assignment.

data-type
The name of an SQL data type. If the name is not qualified, the SQL path is used to perform data type resolution. *data-type* must not specify a distinct type. If a data type has associated attributes, such as length or precision and scale, these attributes should be included when specifying a value for *data-type*. CHAR defaults to a length of 1, and DECIMAL defaults to a precision of 5 and a scale of 0 if not specified. CLOB and DBCLOB default to a length of 1M. When the target data type is XML and the source data type is TIMESTAMP, trailing zeroes in the fractional seconds part of the value are not included in the result. Restrictions on the supported data types are based on the specified cast operand. The default encoding scheme for string data types is Unicode. The encoding scheme can be changed by specifying the CCSID clause.

Table 45. Supported conversions from Non-XML values to XML values

Source data type	Target data type: XML	Resulting XML schema type
DATE	Y	xs:date
TIME	Y	xs:time

Table 45. Supported conversions from Non-XML values to XML values (continued)

Source data type	Target data type: XML	Resulting XML schema type
TIMESTAMP(*p*) WITH TIME ZONE	Y	xs:dateTime

Examples

Example 1: Create a null XML value.

```
XMLCAST(NULL AS XML)
```

Example 2: Convert a value extracted from an XMLQUERY expression into an INTEGER:

```
XMLCAST(XMLQUERY('/PRODUCT/QUANTITY'
  PASSING xmlcol) AS INTEGER)
```

Example 3: Convert a value extracted from an XMLQUERY expression into a varying-length character string:

```
XMLCAST(XMLQUERY('/PRODUCT/NAME'
  PASSING xmlcol) AS VARCHAR(20))
```

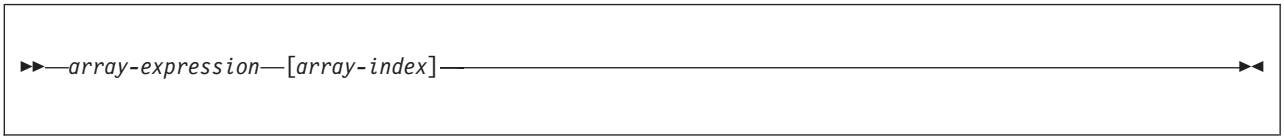
Note that in the above two examples, if the XMLQUERY returns a sequence of more than one node, the XMLCAST specification will return an error.

Example 4: Convert a value extracted from an SQL scalar subquery into an XML value:

```
XMLCAST((SELECT quantity FROM product AS p
  WHERE p.id = 1077) AS XML)
```

Array element specification

The array element specification returns the element from an array specified by *array-index*.



array-expression
Specifies an SQL variable or SQL parameter of an array type, or a CAST specification of a parameter marker to an array type.

[*array-index***]**
An expression that specifies the array index of the element that is to be extracted from the array. An array index value for an ordinary array must be castable to INTEGER. The array index value must be between 1 and the cardinality of the array. An array index value for an associative array must be castable to the data type of the index for the array type. The array index value must represent an element that exists in the array. If the index value is a string that is longer than the index data type, the value is truncated, a warning is issued, and processing continues with the truncated value.

array-index must not be:

- An expression that references the CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP special register
- A nondeterministic function
- A function that is defined with EXTERNAL ACTION
- A function that is defined with MODIFIES SQL DATA
- A sequence expression

The data type of the result is the data type that is specified for the array on the CREATE TYPE (array) statement. If *array-index* is null, or the array is null, the null value is returned.

If the array element is character or graphic data, the CCSID of the result is the CCSID of the array elements of the array type. If the array element is datetime data, the CCSID of the result is 1208.

Examples

Example 1: Suppose that PHONE_NUMBERS is an array variable that is defined as an array type. The array type is defined as an ordinary array of CHAR(10) elements. Also suppose that INT_VAR is an integer variable. The following assignment statements demonstrate how an index for an array element can be specified.

Set the first element of an array to NULL:

```
SET PHONE_NUMBERS[1] = NULL;
```

Set the third element to the value '4164789683':

```
SET PHONE_NUMBERS[3] = '4164789683';
```

Set an array element to '4164788888', and specify the array index with the variable INT_VAR:

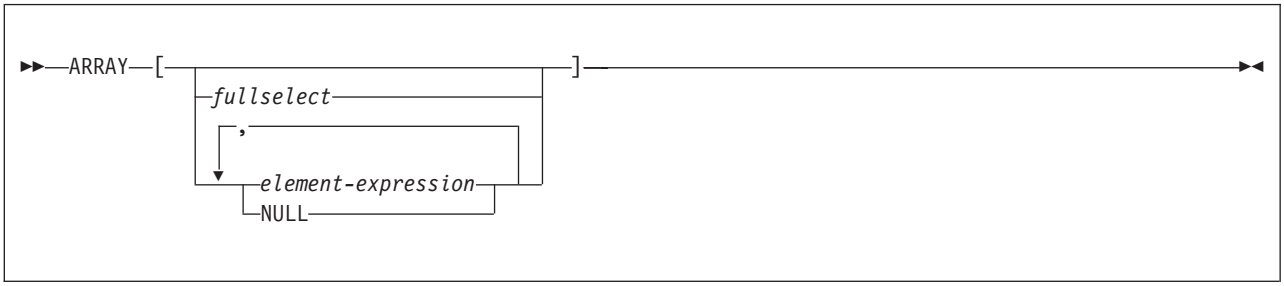
```
SET PHONE_NUMBERS[INT_VAR] = '4164788888';
```

Set an array element to '4164783322', and specify the array index with the expression INT_VAR+5:

```
SET PHONE_NUMBERS[INT_VAR + 5] = '4164783322';
```

Array constructor

An array constructor returns an ordinary array. An array constructor is specified by a list of expressions or a fullselect.



Authorization

No specific authorizations are required to reference an array constructor within an SQL statement. However, for the statement execution to be successful, all other authorization requirements for the statement must be satisfied.

fullselect
A fullselect that returns a single column. The data type of the column must be a data type that can be specified in a CREATE TYPE (array) statement as the data type of an array element. The values that are returned by the fullselect are the elements of the array. The cardinality of the array is equal to the number of rows that are returned by the fullselect. An ORDER BY clause in the fullselect can be used to specify the order among the elements of the array. Otherwise, the order is undefined. The data type of the elements of the resulting array is the same as the data type of the result column of the fullselect.

element-expression
An expression that defines the value of an element in the array. The expression must return a value with a data type that can be specified in a CREATE TYPE (array) statement as the data type of an array element. The cardinality of the array is equal to the number of element expressions. The first element expression is assigned to the array element with array index 1. The second element expression is assigned to the array element with array index 2, and so on. All element expressions must have compatible data types. The data type of the elements of the resulting array are determined based on the rules that are described in "Rules for result data types" on page 144.

NULL
Specifies the null value.

If no value is specified within the brackets, the result is an empty array.

An array constructor cannot be specified in a SELECT list, or in an inline SQL function. An array constructor can only be specified in SQL PL, in an expression as a source value for an assignment.

An array constructor cannot be used to construct an associative array. An associative array can be constructed only by assigning values to individual array elements.

Examples

Example 1: Suppose that the array variable RECENT_CALLS has the array type PHONENUMBERS. Assign an array of fixed numbers to RECENT_CALLS.

```
SET RECENT_CALLS = ARRAY[9055553907, 4165554213, 4085553678];
```

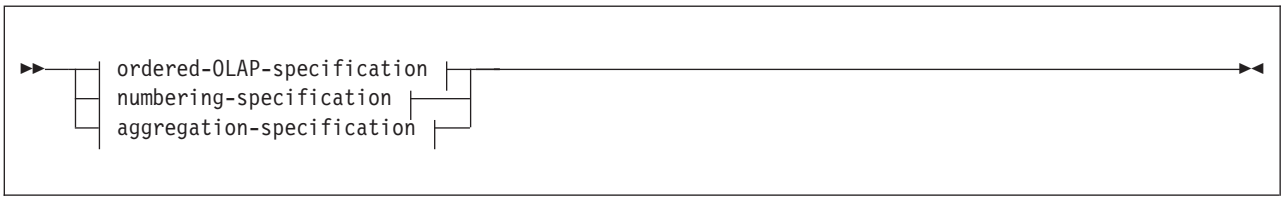
Example 2: Suppose that the array variable DEPT_PHONES has the array type PHONENUMBERS. Assign array phone numbers that are retrieved from the DEPARTMENT_INFO table to DEPT_PHONES.

```
SET DEPT_PHONES =
 ARRAY[SELECT DECIMAL(AREA_CODE CONCAT '555' CONCAT EXTENSION,16)
  FROM DEPARTMENT_INFO
  WHERE DEPTID = 624];
```

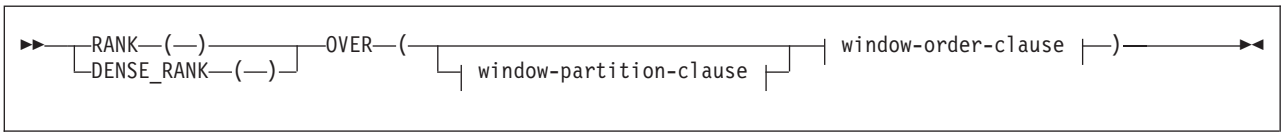
OLAP specification

Online analytical processing (OLAP) specifications provide the ability to return ranking, row numbering, and aggregation information as a scalar value in the result of a query. An OLAP specification can be included in an expression, in a *select-list*, or in the ORDER BY clause of a *select-statement*. The query result to which the OLAP specifications is applied is the result table of the innermost subselect that includes the OLAP specification.

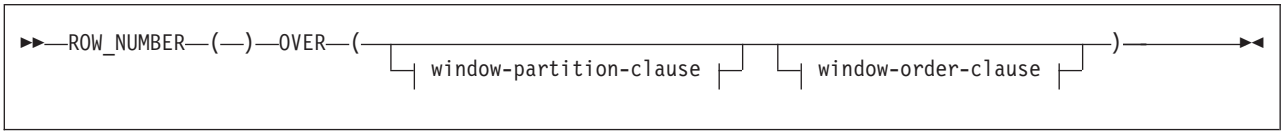
OLAP-specification



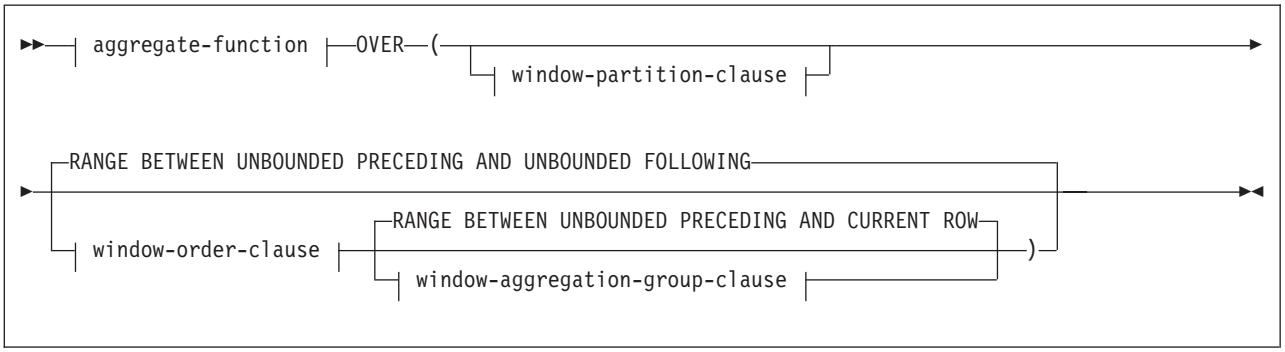
ordered-OLAP-specification



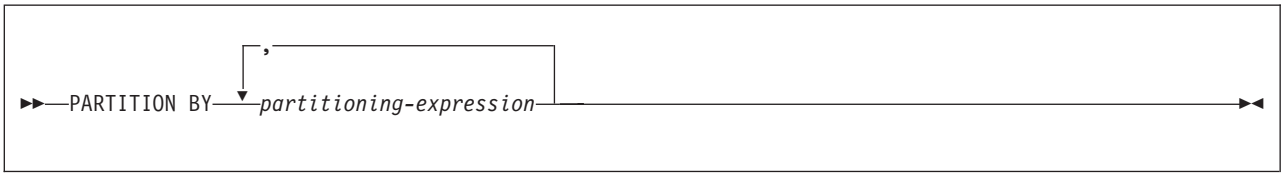
numbering-specification



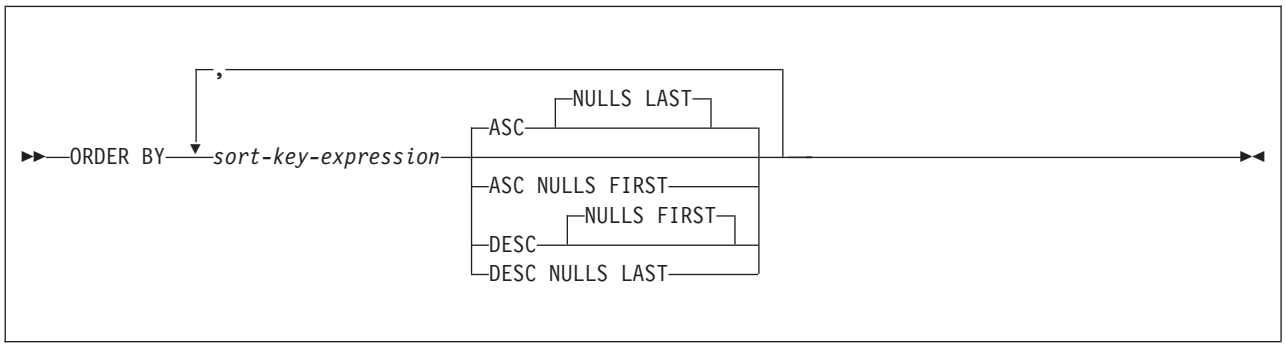
aggregation-specification



window-partition-clause



window-order-clause

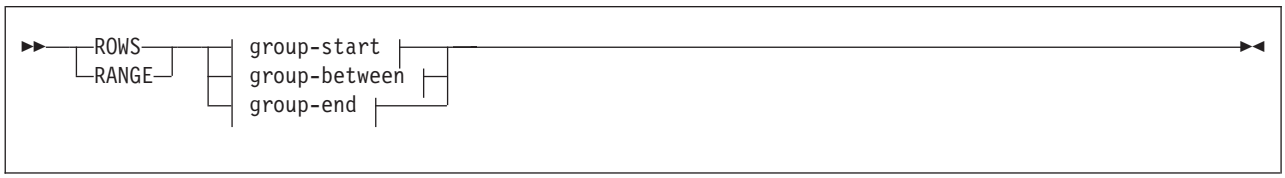


aggregate-function

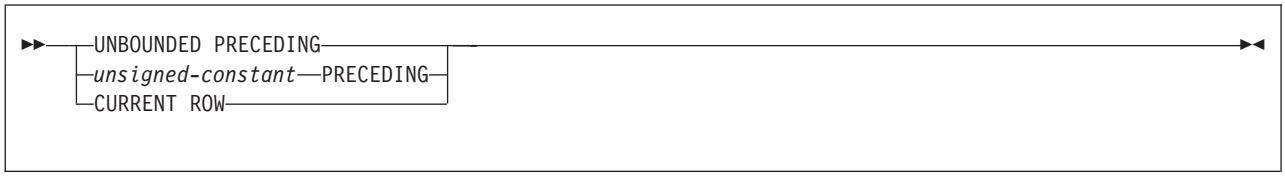
Notes:

1 You cannot specify DISTINCT or ALL for an aggregate function that is included in an aggregation-specification.

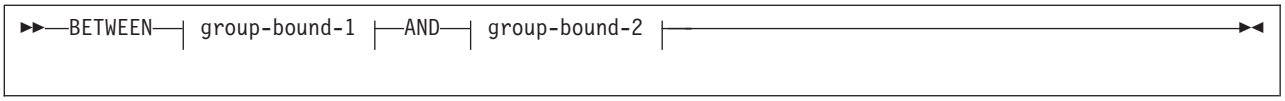
window-aggregation-group-clause



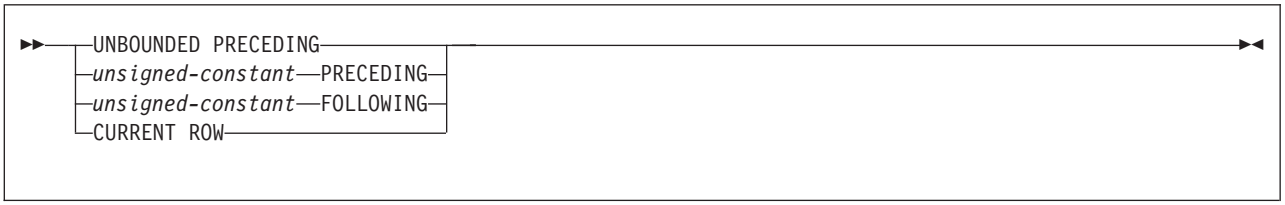
group-start



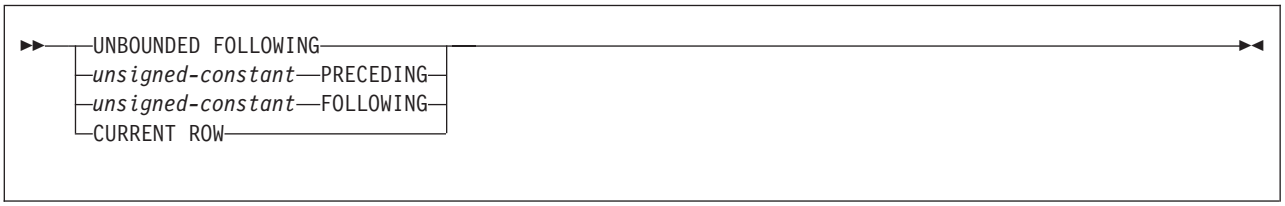
group-between



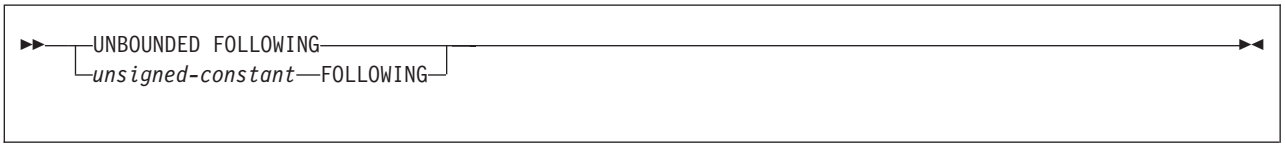
group-bound-1



group-bound-2



group-end



RANK, DENSE_RANK, and ROW_NUMBER are sometimes called window functions.

An OLAP specification is not valid in a **WHERE**, **VALUES**, **GROUP BY**, **HAVING**, or **SET** clause. An OLAP specification cannot be used as an argument of an aggregate function.

When invoking an OLAP specification, a window is specified that defines the rows over which the function is applied and in which order.

The result of a RANK, DENSE_RANK, or ROW_NUMBER specification is BIGINT. The result cannot be null.

RANK or DENSE_RANK
Specifies that the ordinal rank of a row within the specified window is computed. Rows that are not distinct with respect to the ordering within the specified window are assigned the same rank. The results of the ranking can be defined with or without gaps in the numbers that result from duplicate values.

RANK
Specifies that the rank of a row is defined as 1 plus the number of rows that strictly precede the row. Thus, if two or more rows are not distinct with respect to the ordering, there will be one or more gaps in the sequential rank numbering.

DENSE_RANK
Specifies that the rank of a row is defined as 1 plus the number of preceding rows that are distinct with respect to the ordering. Therefore, there will be no gaps in the sequential rank numbering.

ROW_NUMBER
Specifies that a sequential row number is computed for the row that is defined by the ordering, starting with 1 for the first row. If the **ORDER BY** clause is not specified in the window, the row numbers are assigned to the rows in an arbitrary order, as the rows are returned (not according to any **ORDER BY** clause in the *select-statement*).

PARTITION BY (*partitioning-expression***,...)**
Defines the partition within which the OLAP operation is applied. A *partitioning-expression* is an expression that is used in defining the partitioning of the result table. Each column name that is referenced in a *partitioning-expression* must unambiguously reference a column of the result table of the subselect that contains the OLAP specification. A *partitioning-expression* cannot include a *scalar-fullselect* an XMLQUERY or XMLEXISTS expression or any function that is not deterministic or has an external action.

ORDER BY (*sort-key-expression***,...)**
Defines the ordering of rows within a partition that is used to determine the value of the OLAP specification. It does not define the ordering of the result table.

sort-key-expression
Specifies an expression to use in defining the ordering of the rows within a window partition. Each column name that is referenced in a *sort-key-expression* must unambiguously reference a column of the result table of the subselect, including the OLAP specification. A *sort-key-expression* cannot include a scalar fullselect, an XMLQUERY or XMLEXISTS expression, or any function that is not deterministic or that has an external action.

ASC
Specifies that the values of *sort-key-expression* are used in ascending order.

DESC
Specifies that the values of *sort-key-expression* are used in descending order.

NULLS FIRST
Specifies that the window ordering considers null values before all non-null values in the sort order.

NULLS LAST
Specifies that the window ordering considers null values after all non-null values in the sort order.

window-aggregation-group-clause
The aggregation group of a given row is a set of rows that is defined in relation to the given row (in the ordering of the rows in the partition of the given row). *window-aggregation-group-clause* specifies the aggregation group. If this clause is not specified and a *window-order-clause* is also not specified, the aggregation group consists of all rows of the window partition. The aggregation group of all rows of the window partition can be explicitly specified using the RANGE or ROWS clauses.

If *window-order-clause* is specified, but *window-aggregation-group-clause* is not specified, the window aggregation group consists of all rows that precede a

given row of the partition of the given row or all rows that are peers of the given row in the window ordering of the window partition that is defined by the *window-order-clause*

ROW
Specifies that the aggregation group is defined by counting rows.

RANGE
Specifies that the aggregation group is defined by an offset from a sort key.

group-start
Specifies the starting point for the aggregation group. The aggregation group end is the CURRENT ROW. Specifying *group-start* is equivalent to specifying *group-between* as **BETWEEN** *group-start* **AND CURRENT ROW**.

group-between
Specifies that the aggregation group start and end based on either ROWS or RANGE.

group-end
Specifies the ending point for the aggregation group. The aggregation group start is the CURRENT ROW. Specifying *group-end* is equivalent to specifying *group-between* as **BETWEEN CURRENT ROW AND** *group-end*.

UNBOUNDED PRECEDING
Specifies that the entire partition that precedes the current row is included in the aggregation group. This can be specified with either the ROWS or RANGE clauses. Including the entire partition that precedes the current row can also be specified with multiple *sort-key-expressions* in the *window-order-clause*.

UNBOUNDED FOLLOWING
Specifies that the entire partition that follows the current row is included in the aggregation group. This can be specified with either the ROWS or RANGE clauses. Including the entire partition that follows the current row can also be specified with multiple *sort-key-expressions* in the *window-order-clause*.

CURRENT ROW
Specifies that the aggregation group starts or ends based on the current row. IF ROWS is specified, the current row is the aggregation group boundary. If RANGE is specified, the aggregation group boundary includes the set of rows with the values specified for the *sort-key-expression* as the current row. This clause cannot be specified in *group-bound-2* if *group-bound-1* specifies *unsigned-constant* **FOLLOWING**.

unsigned-constant **PRECEDING**
Specifies either the range or the number of rows that precede the current row. If ROWS is specified, *unsigned-constant* must be zero or a positive integer that indicates a number of rows. If RANGE is specified, the data type of *unsigned-constant* must be comparable to the data type of the *sort-key-expression* of the *window-order-clause*. Only one *sort-key-expression* is allowed, and the data type of *sort-key-expression* must allow subtraction. This clause cannot be specified in *group-bound-2* if *group-bound-1* is CURRENT ROW or *unsigned-constant* **FOLLOWING**.

unsigned-constant **FOLLOWING**
Specifies either the range or the number of rows that follow the current row. If ROWS is specified, *unsigned-constant* must be zero or a positive integer that indicates a number of rows. If RANGE is specified, the data type of *unsigned-constant* must be comparable to the data type of the

sort-key-expression of the *window-order-clause*. Only one *sort-key-expression* is allowed, and the data type of *sort-key-expression* must allow addition.

Notes

Using a column mask with an OLAP specification: If a column mask is used to mask the column values in the final result table and an OLAP specification is referenced in the select list that is used to derive the final result table, the column mask cannot be applied to the column that is specified in the *partitioning-expression* or the *sort-key-expression* in the OLAP specification

Syntax alternatives and synonyms: For compatibility, the keywords **DENSERANK** and **ROWNUMBER** can be used as synonyms for **DENSE_RANK** and **ROW_NUMBER** respectively.

Example 1: Display the ranking of employees that have a total salary of more than $30,000, in order by last name:

```
SELECT EMPNO, LASTNAME, FIRSTNME, SALARY+BONUS AS TOTAL_SALARY,
  RANK() OVER(ORDER BY SALARY+BONUS DESC) AS RANK_SALARY
  FROM EMP WHERE SALARY+BONUS > 30000
  ORDER BY LASTNAME;
```

If the result is to be ordered by rank, `ORDER BY LASTNAME` would be replaced with `ORDER BY RANK_SALARY`.

Example 2: Rank the departments according to their average total salary:

```
SELECT WORKDEPT, AVG(SALARY+BONUS) AS AVG_TOTAL_SALARY,
  RANK() OVER(ORDER BY AVG(SALARY+BONUS) DESC) AS RANK_AVG_SAL
  FROM EMP
  GROUP BY WORKDEPT
  ORDER BY RANK_AVG_SAL;
```

Example 3: Rank the departments according to their education level. Having multiple employees with the same rank in the department should not increase the next ranking value:

```
SELECT WORKDEPT, EMPNO, LASTNAME, FIRSTNME, EDLEVEL,
  DENSE_RANK() OVER
   (PARTITION BY WORKDEPT ORDER BY EDLEVEL DESC) AS RANK_EDLEVEL
  FROM EMP
  ORDER BY WORKDEPT, LASTNAME;
```

Example 4: Provide row numbers in the results of a query:

```
SELECT ROW_NUMBER() OVER(ORDER BY WORKDEPT, LASTNAME) AS NUMBER,
  LASTNAME, SALARY
  FROM EMP
  ORDER BY WORKDEPT, LASTNAME;
```

Example 5: List the top five wage earners:

```
SELECT EMPNO, LASTNAME, FIRSTNME, TOTAL_SALARY, RANK_SALARY
  FROM (SELECT EMPNO, LASTNAME, FIRSTNME, SALARY+BONUS AS TOTAL_SALARY,
        RANK() OVER (ORDER BY SALARY+BONUS DESC) AS RANK_SALARY
        FROM EMP) AS RANKED_EMPLOYEE
  WHERE RANK_SALARY < 6
  ORDER BY RANK_SALARY;
```

A nested table expression is used to first compute the result, including the ranking, before the rank can be used in the **WHERE** clause. A common table expression could also have been used.

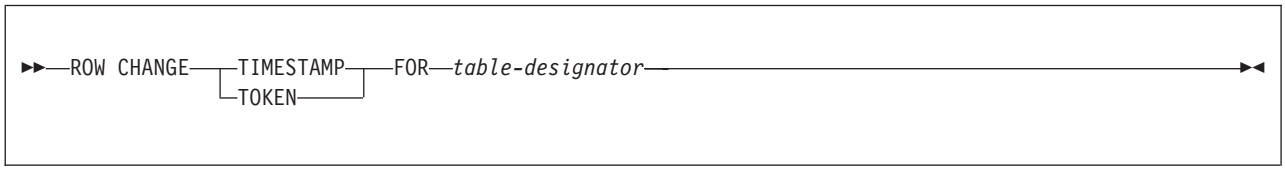
Example 6: The following example is used to calculate the 30 day moving average for the stocks 'ABC' and 'XYX' during 2005:

```
CREATE VIEW V1 AS
 SELECT SYMBOL, TRADINGDATE,
    AVG(CLOSINGPRICE) OVER (PARTITION BY SYMBOL
                ORDER BY TRADINGDATE
                ROWS BETWEEN 29 PRECEDING AND CURRENT ROW)
  FROM DAILYSTOCKDATA
  WHERE SYMBOL IN ('ABC', 'XYZ')
      AND TRADINGDATE BETWEEN DATE('2005-01-01') - 2 MONTHS AND '2005-12-31';

SELECT SYMBOL, TRADINGDATE, MOVINGAVG30DAY
 FROM V1
  WHERE TRADINGDATE BETWEEN '2005-01-01' AND '2005-12-31'
 ORDER BY SYMBOL, TRADINGDATE;
```

ROW CHANGE expression

A ROW CHANGE expression returns a token or a timestamp that represents the last change to a row.

ROW CHANGE expression



TIMESTAMP
Specifies that a timestamp is returned that represents the last time when a row was changed. If the row has not been changed, the result is the time that the initial value was inserted.

TOKEN
Specifies that a token that is a BIGINT value is returned that represents a relative point in the modification sequence of a row. If the row has not been changed, the result is a token that represents when the initial value was inserted.

FOR *table-designator*
Identifies the table in which the expression is referenced. *table-designator* must uniquely identify a base table, a view, or a nested table expression of a subselect. If *table-designator* identifies a view or a nested table expression, the ROW CHANGE expression returns the TIMESTAMP or TOKEN of the base table of the view or the nested table expression. The view or nested table expression must contain only one base table in its outer subselect. *table-designator* must not identify a materialized view, a nested table expression that is materialized, an alias, or a synonym.

The result can be null. The **ROW CHANGE TIMESTAMP** and **ROW CHANGE TOKEN** expressions are not deterministic.

Notes

Tables without a row change timestamp column:
For tables without a row change timestamp column, the ROW CHANGE TIMESTAMP expression returns a timestamp value that reflects changes made to the page instead of to the row. This timestamp value indicates that at least one row in the page has changed, but does not indicate which row, or even how many rows, have changed. The ROW CHANGE TIMESTAMP expression might indicate that a row has changed, however, the change might be for other rows in the same page.

In a data sharing environment, the returned timestamp value is based on the LRSN value of the page and reflects the most recent time the page was modified.

In a non-data sharing environment, the returned timestamp value is based on the RBA value of the page. In a non-data sharing environment, changes made to the same page within a half hour of each other might be indistinguishable. For example, issuing the following SELECT statements in a non-data sharing environment will possibly return the same value, even though the row was changed between the two SELECT statements:

```
CREATE TABLE T1 (C1 INTEGER NOT NULL);
INSERT INTO T1 VALUES (1);
SELECT ROW CHANGE TIMESTAMP FOR T1 FROM T1;
UPDATE T1 SET C1 = 2 WHERE C1 = 1;
SELECT ROW CHANGE TIMESTAMP FOR T1 FROM T1;
```

Example 1:

The following example returns all the rows that have been changed in the last day:

```
SELECT * FROM ORDERS
  WHERE ROW CHANGE TIMESTAMP FOR ORDERS >
    CURRENT TIMESTAMP - 24 HOURS;
```

Example 2:

The following example returns a timestamp value that corresponds to the most recent change to each row from the EMP table for those employees in department 20:

```
SELECT ROW CHANGE TIMESTAMP FOR EMP
  FROM EMP WHERE DEPTNO = 20;
```

Example 3:

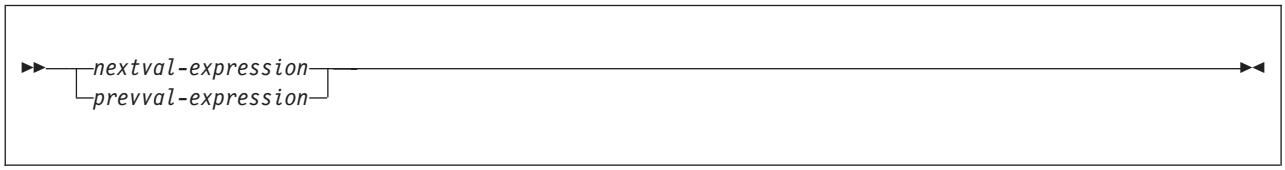
The following example returns a BIGINT value that corresponds to a relative point in the modification sequence of EMP with employee number '3500':

```
SELECT ROW CHANGE TOKEN FOR EMP
  FROM EMP WHERE EMPNO = '3500';
```

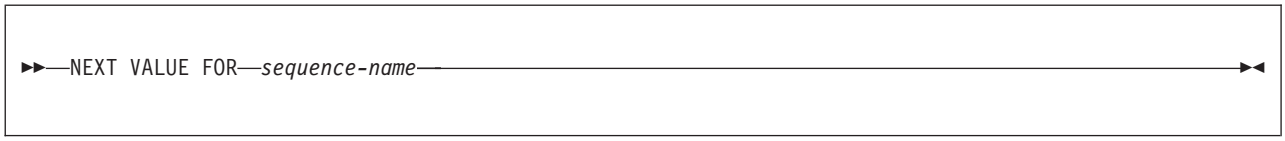
Sequence reference

A sequence is referenced by using the NEXT VALUE and PREVIOUS VALUE expressions specifying the name of the sequence.

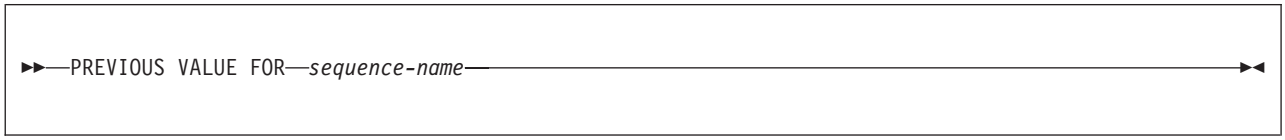
sequence-reference



nextval-expression



prevval-expression



nextval-expression
A NEXT VALUE expression generates and returns the next value for a specified sequence. A new value is generated for a sequence when a NEXT VALUE expression specifies the name of the sequence. However, if there are multiple instances of a NEXT VALUE expression specifying the same sequence name within a query, the sequence value is incremented only once for each row of the result, and all instances of **NEXT VALUE** return the same value for a row of the result. The **NEXT VALUE** expression is a not deterministic with external actions since it causes the sequence value to be incremented.

When the next value for the sequence is generated, if the maximum value for an ascending sequence or the minimum value for a descending sequence of the logical range of the sequence is exceeded and the **NO CYCLE** option is in effect, then an error occurs. To avoid this error, either alter the sequence attributes to extend the range of value or to enable cycles for the sequence or drop and re-create the sequence with a different data type that allows a larger range of values.

The data type and length attributes of the result of a NEXT VALUE expression are the same as for the specified sequence. The result cannot be null.

prevval-expression
A PREVIOUS VALUE expression returns the most recently generated value for the specified sequence for a previous statement within the current application process. This value can be repeatedly referenced by using PREVIOUS VALUE expressions to specify the name of the sequence. There can be multiple instances of PREVIOUS VALUE expressions specifying the same sequence name within a single statement and they all return the same value.

A PREVIOUS VALUE expression can be used only if a NEXT VALUE expression specifying the same sequence name has already been referenced in the current application process.

The data type and length attributes of the result of a PREVIOUS VALUE expression are the same as for the specified sequence. The result cannot be null.

sequence-name
Identifies the sequence that is to be referenced. The combination of name and the implicit or explicit schema name must identify an existing sequence at the current server. *sequence-name* must not be the name of an internal sequence object that is generated by DB2 for an identity column. The contents of the SQL PATH are not used to determine the implicit qualifier of a sequence name.

Authorization:
If a sequence is referenced in a statement, the privileges that are held by the authorization ID of the statement must include at least one of the following:

- For the sequence identified in the statement:
  - The USAGE privilege on the sequence
  - Ownership of the sequence
- SYSADM or SYSCTRL authority

Generating values with NEXT VALUE:
When a value is generated for a sequence, that value is consumed, and the next time that a value is requested, a new value will be generated. This is true even when the statement containing the NEXT VALUE expression fails or is rolled back.

Scope of NEXT VALUE and PREVIOUS VALUE:
The value of PREVIOUS VALUE cannot be directly set and is a result of executing the NEXT VALUE expression for the sequence. The value of PREVIOUS VALUE persists until the next value is generated for the sequence in the current session, the sequence is dropped or altered, or the application session ends.

The value for the sequence cannot persist across a COMMIT or ROLLBACK for a local or remote application if, after the COMMIT or ROLLBACK, the DB2 application thread or server thread is assigned to another user or DB2 connection because of some form of thread reuse, re-signon, or connection pooling is in effect. For example, this can occur for CICS-DB2 applications and for client applications or middleware products that save the state of a session and then restore the state of a session for subsequent processing because they are not able to restore the NEXT or PREVIOUS VALUES for a sequence. In these situations, the availability of the value for a sequence should only be relied on until the end of the transaction. Examples of where this type of situation can occur include applications that do the following:

- issue an EXEC CICS SYNCPOINT command
- use XA protocols
- use connection pooling
- use the connection concentrator
- use Sysplex workload balancing
- connect to a z/OS server that uses DDF inactive threads

When there is a need to preserve the value that is associated with NEXT VALUE or PREVIOUS VALUE expressions across transaction boundaries for local or distributed applications that are subject to thread reuse, re-signon, or connection pooling, take one of the following actions to prevent the local or server thread from re-signon, being reused by a different user, or from being pooled:

- Define at least one cursor as WITH HOLD and leave it as OPEN.
- Specify the bind option KEEPDYNAMIC(YES).

Use as a unique key value:

The same sequence number can be used as a unique key value in two separate tables by referencing the sequence number with a NEXT VALUE expression for the first row (this generates the sequence value), and a PREVIOUS VALUE expression for the other rows (the instance of PREVIOUS VALUE refers to the sequence value most recently generated in the current session), as shown in the following example:

```
INSERT INTO ORDER (ORDERNO, CUSTNO)
  VALUES (NEXT VALUE FOR ORDER_SEQ, 123456);
INSERT INTO LINE_ITEM (ORDERNO, PARTNO, QUANTITY)
  VALUES (PREVIOUS VALUE FOR ORDER_SEQ, 987654, 1);
```

Allowed use of NEXT VALUE and PREVIOUS VALUE:

The NEXT VALUE and PREVIOUS VALUE expressions can be specified in the following places:

- Within the *select-clause* of a SELECT statement or SELECT statement that does not contain a **DISTINCT** keyword, a **GROUP BY** clause, an **ORDER BY** clause, or a set operator.
- Within a **VALUES** clause of an INSERT statement, including a multiple row INSERT statement with multiple **VALUES** clauses and the insert operation of a MERGE statement, which can include a NEXT VALUE expression for a particular sequence name for each **VALUES** clause.
- Within the *select-clause* of the fullselect of an INSERT statement.
- Within the **SET** clause of a searched or positioned UPDATE statement, including the update operation of the MERGE statement, though NEXT VALUE cannot be specified in the *select-clause* of the fullselect of an expression in the **SET** clause.

  A PREVIOUS VALUE expression can be specified anywhere with a **SET** clause of an update operation (the UPDATE or MERGE statement), but a NEXT VALUE expression can be specified only in a **SET** clause if it is not within the *select-clause* of the fullselect of an expression. For instance, the following uses of sequence references are supported:

  ```
  UPDATE T SET C1 = (SELECT PREVIOUS VALUE FOR S1 FROM T);
  UPDATE T SET C1 = PREVIOUS VALUE FOR S1;
  UPDATE T SET C1 = NEXT VALUE FOR S1;
  ```

  The following uses of sequence references are not supported:

  ```
  UPDATE T SET C1 = (SELECT NEXT VALUE FOR S1 FROM T);
  SET :C2 = (SELECT NEXT VALUE FOR S1 FROM T);
  ```

- In a **SET** *host-variable* or *assignment-statement*, except within the *select-clause* of the fullselect of an expression.

  The following uses of sequence references are supported:

  ```
  SET ORDERNUM = NEXT VALUE FOR INVOICE;
  SET ORDERNUM = PREVIOUS VALUE FOR INVOICE;
  ```

  The following uses of sequence references are not supported:

```
SET X = (SELECT NEXT VALUE FOR S1 FROM T);
SET X = (SELECT PREVIOUS VALUE FOR S1 FROM T);
```

- In a **VALUES** or **VALUES INTO** statement though not within the *select-clause* of the fullselect of an expression.
- Within the *SQL-routine-body* of a CREATE or ALTER PROCEDURE statement for a SQL procedure.
- Within the *RETURN-statement* of a CREATE FUNCTION statement for an SQL function.
- Within the *SQL-trigger-body* of a CREATE TRIGGER statement (PREVIOUS VALUE is not allowed).

Use of PREVIOUS VALUE in a nested application:

PREVIOUS VALUE is defined to have a linear scope within an application session. Therefore, in a nested application on entry to a nested function, procedure, or trigger, the nested application inherits the most recently generated value for a sequence. That is, an invocation of PREVIOUS VALUE in a nested application reflects sequence activity done in the invoking environment prior to entering the nested application. In addition, on return from a function, procedure, or trigger, the invoking application is affected by any sequence activity in the lower level applications. That is, an invocation of PREVIOUS VALUE in the invoking application after returning from the nested application reflects any sequence activity that occurred in the lower level applications.

Restrictions on the use of NEXT VALUE and PREVIOUS VALUE:

Some of the places where the NEXT VALUE and PREVIOUS VALUE expressions cannot be specified include the following:

- Join condition of a full outer join
- DEFAULT value for a column in a CREATE TABLE or ALTER TABLE statement
- Materialized query table definition in a CREATE TABLE or ALTER TABLE statement
- Condition of a CHECK constraint
- Input value specification for LOAD
- CREATE VIEW statement
- The SELECT list of a subselect that contains a NOT ATOMIC data change statement
- ORDER BY clause when used in an OLAP specification

In addition, the NEXT VALUE expression cannot be specified in the following places:

- CASE expression
- Parameter list of an aggregate function
- Subquery in a context other than those explicitly allowed
- SELECT statement for which the outer SELECT contains a DISTINCT operator or a GROUP BY clause
- SELECT statement for which the outer SELECT is combined with another SELECT statement using a set operator
- Join condition of a join
- Nested table expression
- Parameter list of a table function

- *select-clause* of the fullselect of an expression in the SET clause of an UPDATE, a DELETE, or a MERGE statement.
- WHERE clause of the outer-most SELECT statement or a DELETE, , an UPDATE, or a MERGE statement
- ORDER BY clause of the outer-most SELECT statement
- IF, WHILE, DO UNTIL, or CASE statements in an SQL routine

Using sequence expressions with a cursor:

Normally, a `SELECT NEXT VALUE FOR ORDER_SEQ FROM T1` would produce a result table containing as many generated values from the sequence ORDER_SEQ as the number of rows retrieved from T1. A reference to a NEXT VALUE expression in the SELECT statement of a cursor refers to a value that is generated for a row of the result table. A sequence value is generated for a NEXT VALUE expression each time a row is retrieved.

If blocking is done at a client in a DRDA environment, sequence values might get generated at the DB2 server before the processing of an application's FETCH statement. If the client application does not explicitly fetch all the rows that have been retrieved from the database, the application will never see all those values of the sequence that are generated but not fetched (as many values as the rows that are not fetched). These generated but not fetched values might constitute a gap in the sequence. If it is important to prevent such a gap in the sequence, do the following:

- Use NEXT VALUE only where it would function without being controlled by a cursor and where block-fetching by the client will have no effect on it.
- If you must use NEXT VALUE in the SELECT statement of a cursor-definition, weigh the importance of preventing the gap against performance and other implications of taking the following actions:
  - Use FETCH FOR 1 ROW ONLY clause with the SELECT statement.
  - Try preventing block-fetch by other means documented in *DB2 Application Programming and SQL Guide*,

Using the PREVIOUS VALUE expression with a cursor:

A reference to the PREVIOUS VALUE expression in a SELECT statement of a cursor is evaluated at OPEN time. In other words, a reference to the PREVIOUS VALUE expression in the SELECT statement of a cursor refers to the last value generated by this application process for the specified sequence prior to the opening of the cursor and, once evaluated at OPEN time, the value returned by PREVIOUS VALUE within the select statement of the cursor will not change from FETCH to FETCH, even if NEXT VALUE is invoked with the select statement of the cursor. After the cursor is closed, the value of PREVIOUS VALUE will be the last NEXT VALUE that is generated by the application process.

IF PREVIOUS VALUE is used in the SELECT statement of a cursor while the cursor is open, the value of PREVIOUS VALUE would be the last NEXT VALUE for the generated sequence before the cursor was opened. After the cursor is closed, the value of PREVIOUS VALUE would be the last NEXT VALUE generated by the application process.

Syntax alternatives and synonyms:

For compatibility, the keywords NEXTVAL and PREVVAL can be used as synonyms for NEXT VALUE and PREVIOUS VALUE respectively.

sequence-name.NEXTVAL can be specified in place of **NEXT VALUE FOR** *sequence-name*, and *sequence-name*.CURRVAL can be specified in place of **PREVIOUS VALUE FOR** *sequence-name*.

Example

Assume that there is a table called ORDER, and that a sequence called ORDER_SEQ is created as follows:

```
CREATE SEQUENCE ORDER_SEQ START WITH 1
                       INCREMENT BY 1
                       NO MAXVALUE
                       NO CYCLE
                       CACHE 24
```

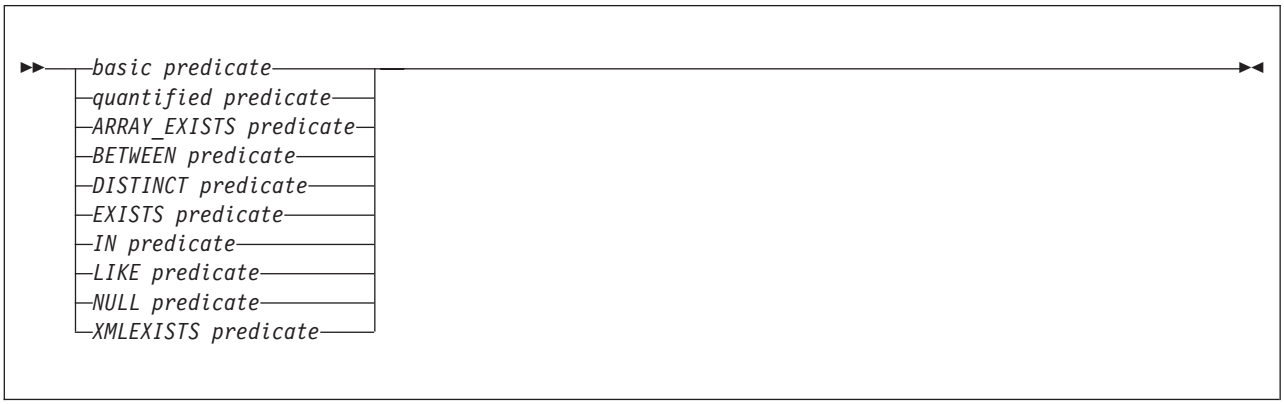
The following examples illustrate how to generate an ORDER_SEQ sequence number with a NEXT VALUE expression:

```
INSERT INTO ORDER (ORDERNO, CUSTNO)
        VALUES (NEXT VALUE FOR ORDER_SEQ, 123456);
UPDATE ORDER SET ORDERNO = NEXT VALUE FOR ORDER_SEQ
    WHERE CUSTNO = 123456;
VALUES NEXT VALUE FOR ORDER_SEQ INTO :HV_SEQ;
```

Predicates

A *predicate* specifies a condition that is true, false, or unknown about a given row or group.

The types of predicates are:



The following rules apply to predicates of any type:

- Predicates are evaluated after the expressions that are operands of the predicate.
- All values that are specified in the same predicate must be compatible.
- Except for the EXISTS predicate, a subquery in a predicate must specify a single column unless the operand on the other side of the comparison operator is a fullselect.
- The value of a host variable can be null (that is, the variable can have a negative indicator variable).
- The CCSID conversion of operands of predicates that involve two or more operands is done according to"Conversion rules for comparisons" on page 138.
- Use of an XML value is limited to the NULL or XMLEXISTS predicates.

Row-value-expression: The operand of several predicates (basic, quantified, DISTINCT, and IN) can be a *row-value-expression*:

A *row-value-expression* returns a single row that consists of one or more column values. The values can be specified as a list of expressions. The number of columns that are returned by the *row-value-expression* is equal to the number of expressions that are specified in the list.

Other predicate examples: In addition to the examples of predicates in the following topics, see information on distinct type comparisons in "Assignment and comparison" on page 121, which contains several examples of predicates that use distinct types."Distinct type comparisons" on page 142, which contains several examples of predicates that use distinct types.

Related concepts:

Predicates and access path selection (DB2 Performance)

Related tasks:

Using predicates efficiently (DB2 Performance)

Writing efficient SQL queries (DB2 Performance)

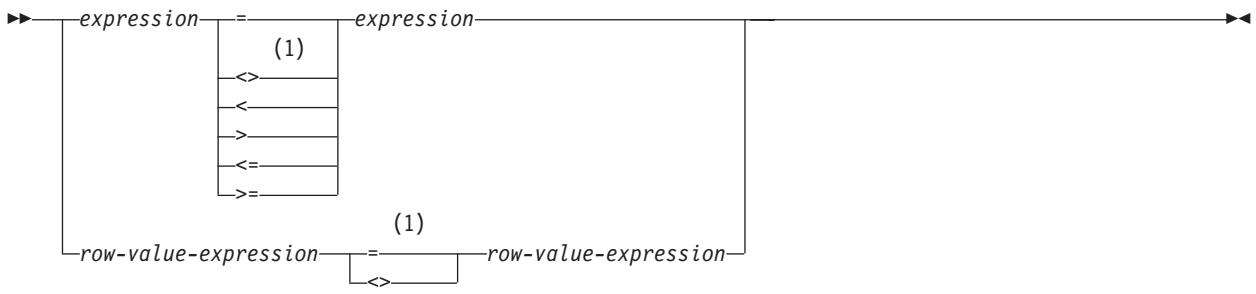
Related reference:

"where-clause" on page 795

"having-clause" on page 799

Basic predicate

A *basic predicate* compares two values or compares a set of values with another set of values.



Notes:

1 Other comparison operators are also supported.[15]

When *expression* is a fullselect, the fullselect must return a single result column with a single value, whether null or not null. If the value of either operand is null or the result of the fullselect is empty, the result of the predicate is unknown. Otherwise, the result is either true or false.

When a *row-value-expression* is specified on the left side of the operator (= or <>), another *row-value-expression*, with an identical number of value expressions, must be specified on the right side. The data types of the corresponding expressions or columns of the *row-value-expressions* must be compatible. The value of each expression on the left side is compared with the value of its corresponding expression on the right side. The result of the predicate depends on the operator, as in the following two cases:

- If the operator is =, the result of the predicate is:
  - True if all pairs of corresponding value expressions evaluate to true.
  - False if any one pair of corresponding value expressions evaluates to false.
  - Otherwise, unknown (that is, if at least one comparison of corresponding value expressions is unknown because of a null value and no pair of corresponding value expressions evaluates to false).
- If the operator is <>, the result of the predicate (x1,x2,...,xn) <> (y1,y2,...,yn) is:
  - True, if and only if xi=yi evaluates to false for some value of i. (that is, there is at least one pair of non-null values, xi and yi, that are not equal to each other)

15. The following forms of the comparison operators are also supported in basic and quantified predicates in code pages where the exclamation point is X'5A': !=, !<, and !> . In addition, the forms ¬=, ¬<, and ¬> are supported as long as the codepoint used for the logical not symbol is the correct one for the specified code page. These forms of the operators are intended only to support existing SQL statements that use them and are not recommended for use when writing new SQL statements.

    A logical not sign (¬) can cause parsing errors in statements passed from one DBMS to another. The problem occurs if the statement undergoes character conversion with certain combinations of source and target CCSIDs. To avoid this problem, substitute an equivalent operator for any operator that includes a not sign. For example, substitute '<>' for '¬=', '<=' for '¬>', and '>=' for '¬<'.

– False, if and only if xi=yi evaluates to true for every value of i. (that is, (x1,x2,...,xn)=(y1,y2,...,yn) is true)
– Otherwise, unknown (that is, xi or yi is a null value for some value of i, and there is no value of j such that xj=yj evaluates to false).

Table 46. For values x and y

Predicate	**Is true if and only if ...**
x = *y*	*x* is equal to *y*
x <> *y*	*x* is not equal to *y*
x < *y*	*x* is less than *y*
x > *y*	*x* is greater than *y*
x <= *y*	*x* is less than or equal to *y*
x >= *y*	*x* is greater than or equal to *y*

Examples for values x and y:

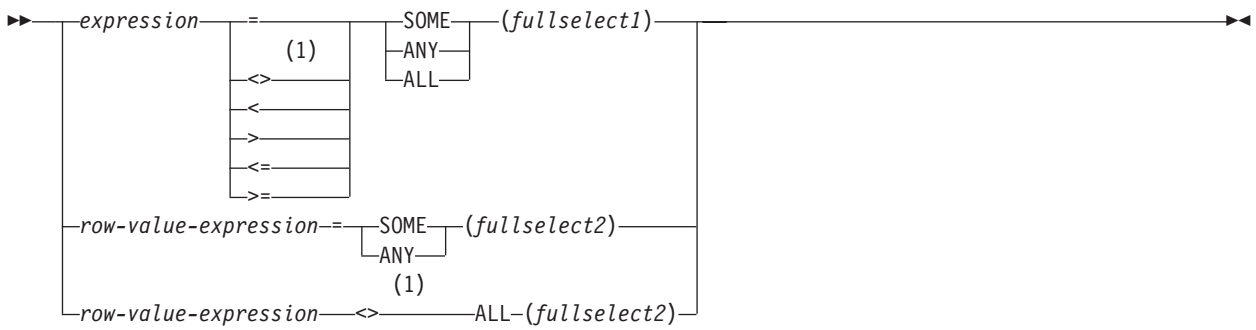
```
EMPNO = '528671'
SALARY < 20000
PRSTAFF <> :VAR1
SALARY >=  (SELECT AVG(SALARY) FROM DSN8B10.EMP)
```

Example: List the name, first name, and salary of the employee who is responsible for the 'SECRET' project. This employee might appear in either the PROJA1 or PROJA2 tables. A UNION is used in case the employee appears in both tables to eliminate duplicate RESPEMP values.

```
SELECT LASTNAME, FIRSTNAME, SALARY
  FROM DSN8B10.EMP X
  WHERE EMPNO = (
SELECT RESPEMP
  FROM PROJA1 Y
  WHERE MAJPROJ = 'SECRET'
UNION
SELECT RESPEMP
  FROM PROJA2 Z
  WHERE MAJPROJ = 'SECRET');
```

Quantified predicate

A *quantified predicate* compares a value or values with a collection of values.



Notes:

1 Other comparison operators are also supported.[15]

When *expression* is specified, *fullselect1* must return a single result column, and can return any number of values, whether null or not null. The result depends on the operator that is specified:

- When the operator is ALL, the result of the predicate is:
  - True – if the result of the fullselect is empty or if the specified relationship is true for every value returned by the fullselect.
  - False – if the specified relationship is false for at least one value returned by the fullselect.
  - Unknown – if the specified relationship is not false for any values returned by the fullselect and at least one comparison is unknown because of a null value.
- When the operator is SOME or ANY, the result of the predicate is:
  - True – if the specified relationship is true for at least one value returned by the fullselect.
  - False – if the result of the fullselect is empty or if the specified relationship is false for every value returned by the fullselect.
  - Unknown – if the specified relationship is not true for any of the values returned by the fullselect and at least one comparison is unknown because of a null value.

When *row-value-expression* is specified, the number of result columns returned by *fullselect2* must be the same as the number of value expressions specified by *row-value-expression*, and *fullselect2* can return any number of rows of values. The data types of the corresponding expressions of the row value expressions must be compatible. The value of each expression from *row-value-expression* is compared with the value of the corresponding result column from *fullselect2*. The value of the predicate depends on the operator that is specified:

- When the operator is ALL, the result of the predicate is:
  - True – if the result of *fullselect2* is empty or if the specified relationship is true for every row returned by *fullselect2*.
  - False – if the specified relationship is false for at least one row returned by *fullselect2*.

  - Unknown – if the specified relationship is not false for any row returned by *fullselect2* and at least one comparison is unknown because of a null value.
- When the operator is SOME or ANY, the result of the predicate is:
  - True – if the specified relationship is true for at least one row returned by *fullselect2*
  - False – if the result of the fullselect is empty or if the specified relationship is false for every row returned by *fullselect2*.
  - Unknown – if the specified relationship is not true for any of the rows returned by *fullselect2* and at least one comparison is unknown because of a null value.

Quantified predicates are equivalent to IN predicates. See Table 52 on page 310 for some examples of equivalent quantified and IN predicates.

Examples: Use the following tables when referring to the following examples. In all examples, "row *n* of TBLA" refers to the row in TBLA for which COLA has the value *n*.

Table 47. TBLA

COLA
1
2
3
4

Table 48. TBLB

COLB	COLC
2	2
3	– –

Table 49. TBLC

COLB	COLC
2	2

Example 1: In the following predicate, the fullselect returns the values 2 and 3. The predicate is false for rows 1, 2, and 3 of TBLA, and is true for row 4.

```
COLA > ALL(SELECT COLB FROM TBLB
       UNION
       SELECT COLB FROM TBLC)
```

Example 2: In the following predicate, the fullselect returns the values 2 and 3. The predicate is false for rows 1 and 2 of TBLA, and is true for rows 3 and 4.

```
COLA > ANY(SELECT COLB FROM TBLB
       UNION
       SELECT COLB FROM TBLC)
```

Example 3: In the following predicate, the fullselect returns the values 2 and null. The predicate is false for rows 1 and 2 of TBLA, and is unknown for rows 3 and 4. The result is an empty table.

```
COLA > ALL(SELECT COLC FROM TBLB
       UNION
       SELECT COLC FROM TBLC)
```

Example 4: In the following predicate, the fullselect returns the values 2 and null. The predicate is unknown for rows 1 and 2 of TBLA, and is true for rows 3 and 4.

```
COLA > SOME(SELECT COLC FROM TBLB
       UNION
       SELECT COLC FROM TBLC)
```

Example 5: In the following predicate, the fullselect returns an empty result column. Hence, the predicate is true for all rows of TBLA.

```
COLA < ALL(SELECT COLB FROM TBLB WHERE COLB>3
       UNION
       SELECT COLB FROM TBLC WHERE COLB>3)
```

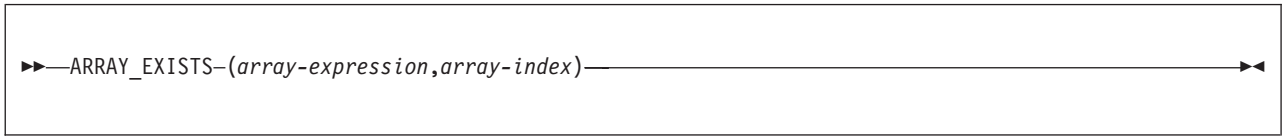
Example 6: In the following predicate, the fullselect returns an empty result column. Hence, the predicate is false for all rows of TBLA.

```
COLA < ANY(SELECT COLB FROM TBLB WHERE COLB>3
       UNION
       SELECT COLB FROM TBLC WHERE COLB>3)
```

If COLA were null in one or more rows of TBLA, the predicate would still be false for all rows of TBLA.

ARRAY_EXISTS predicate

The ARRAY_EXISTS predicate tests for the existence of an array element with the specified index in an array.



array-expression

Specifies one of the following items:

- An SQL variable or SQL parameter of an array type
- A CAST specification of an array or parameter marker to an array type.

array-index

Specifies the index for the array element that is to be tested. An array index value for an ordinary array must be castable to INTEGER. An array index value for an associative array must be castable to the data type of the array index.

array-index must *not* be an expression that references any of the following items:

- The CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP special register
- A nondeterministic function
- A function that is defined with EXTERNAL ACTION
- A function that is defined with MODIFIES SQL DATA
- A sequence expression

The result of the ARRAY_EXISTS predicate is:

- True if *array-variable* includes an array index that is equal to the result of casting *array-index* to the data type of the array index of *array-variable*.
- False under either of the following conditions:
  - *array-variable* does not include an array index that is equal to the result of casting *array-index* to the data type of the array index of *array-variable*.
  - Either argument is null.
- Cannot be unknown.

Example: Suppose that array variable RECENT_CALLS is defined as an ordinary array of array type PHONENUMBERS. The following IF statement tests whether the recent calls list has reached the 40th saved call. If it has, the local Boolean variable EIGHTY_PERCENT is set to true:

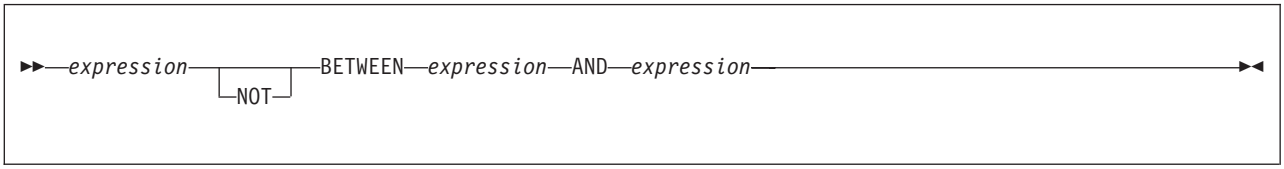
```
IF (ARRAY_EXISTS(RECENT_CALLS, 40))
 THEN SET EIGHTY_PERCENT = TRUE;
END IF
```

Related concepts:

"Array types" on page 108

BETWEEN predicate

The BETWEEN predicate determines whether a given value lies between two other given values that are specified in ascending order.



Each of the predicate's two forms has an equivalent search condition, as shown in the following table:

Table 50. BETWEEN predicate and equivalent search conditions

BETWEEN predicate	Equivalent search condition
value1 BETWEEN *value2* AND *value3*	*value1* >= *value2* AND *value1* <= *value3*[1]
value1 NOT BETWEEN *value2* AND *value3* or, equivalently: NOT(*value1* BETWEEN *value2* AND *value3*)	*value1* < *value2* OR *value1* > *value3*[1]

Note: 1. Might not be equivalent if *value1*, *value2*, or *value3* are columns or derived values based on columns that are not the same CCSID set because the clause is evaluated in Unicode.

Search conditions are discussed in "Search conditions" on page 324.

If the operands include a mixture of datetime values and valid string representations of datetime values, all values are converted to the data type of the datetime operand.

Example: Consider the following predicate:

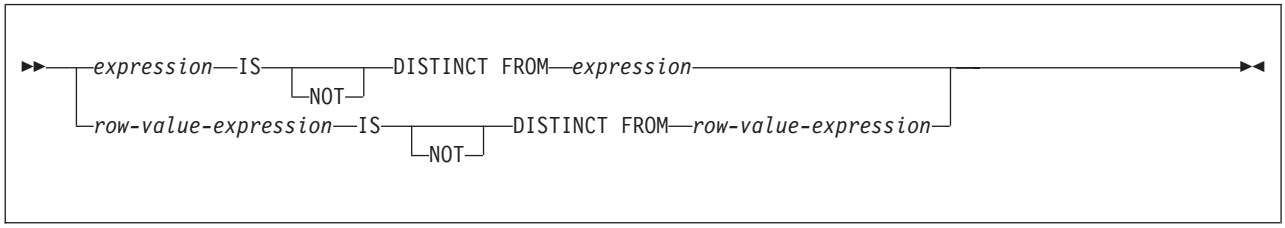
```
A BETWEEN B AND C
```

The following table shows the value of the predicate for various values of *A*, *B*, and *C*.

Value of *A*	Value of *B*	Value of *C*	Value of predicate
1,2, or 3	1	3	true
0 or 4	1	3	false
0	1	null	false
4	null	3	false
null	any value	any value	unknown
2	1	null	unknown
3	null	4	unknown

DISTINCT predicate

A distinct predicate compares a value with another value or a set of values with another set of values.



expression and *row-value-expression* cannot be array expressions.

The number of elements that are returned by the *row-value-expression* that specified after the distinct operator must match the number of elements that are returned by the *row-value-expression* that is specified prior to the distinct operator. The data types of the corresponding columns or expressions of the *row-value-expressions* must be compatible. When the predicate is evaluated, the value of each expression on the left side is compared with the value of its corresponding expression on the right side. The result of the predicate depends on the form of the predicate.

When the predicate is **IS DISTINCT**, the result of the predicate is true if at least one comparison of a pair of corresponding value expressions evaluates to false. Otherwise, the result of the predicate is false. The result cannot be unknown.

When the predicate **IS NOT DISTINCT FROM**, the result of the predicate is true if all pairs of corresponding value expressions evaluate to true (null values are considered equal to null values). Otherwise, the predicate is false. The result cannot be unknown.

The DISTINCT predicate cannot be used in the following contexts:
- The **ON** *join-condition* of a full outer join
- A check constraint
- A quantified predicate

The following DISTINCT predicates are logically equivalent to the corresponding search conditions:

Table 51. DISTINCT predicates and logically equivalent search conditions

DISTINCT predicate	Search condition

`value 1 IS NOT DISTINCT FROM value2`	`(value1 IS NOT NULL`
	` AND value2 IS NOT NULL`
	` AND value1 = value 2)`
	`OR`
	`(value1 IS NULL`
	` AND value2 IS NULL)`

`value 1 IS DISTINCT FROM value2`	`NOT (value1 IS NOT DISTINCT FROM value2)`

Example 1: Assume that T1 is a single-column table with three rows. Column C1 has the following values: 1, 2, and null. Consider the following query:

```
SELECT * FROM T1
   WHERE C1 IS DISTINCT FROM :HV;
```

The following table shows the value of the predicate for various values of C1 and the host variable.

Value of C1	**Value of *HV***	**Result of predicate**
1	2	True
2	2	False
null	2	True
1	null	True
2	null	True
null	null	False

Example 2: Assume the same table as in the first example, but now consider the negative form of the predicate in the query:

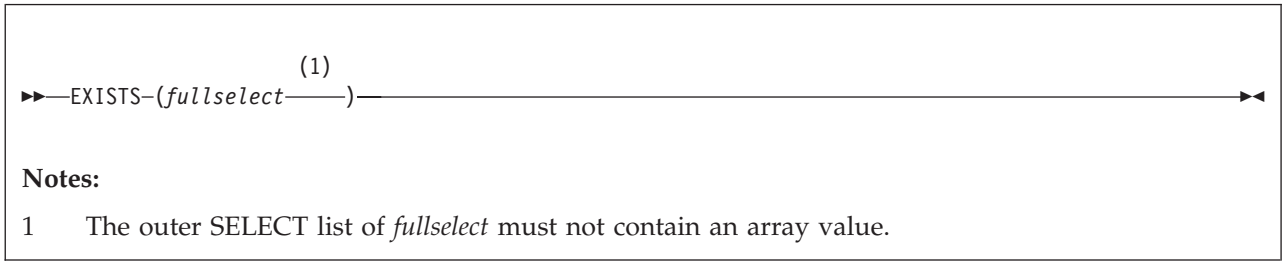
```
SELECT * FROM T1
   WHERE C1 IS NOT DISTINCT FROM :HV;
```

The following table shows the value of the predicate for various values of C1 and the host variable.

Value of C1	**Value of *HV***	**Result of predicate**
1	2	False
2	2	True
null	2	False
1	null	False
2	null	False
null	null	True

EXISTS predicate

The EXISTS predicate tests for the existence of certain rows. The fullselect can specify any number of columns, and can result in true or false.



Notes:

1 The outer SELECT list of *fullselect* must not contain an array value.

The result of the EXISTS predicate:

- Is true only if the number of rows that is specified by the fullselect is not zero.
- Is false only if the number of rows specified by the fullselect is zero.
- Cannot be unknown.

The SELECT clause in the fullselect can specify any number of columns because the values returned by the fullselect are ignored. For convenience, use:

```
SELECT *
```

Unlike the NULL, LIKE, and IN predicates, the EXISTS predicate has no form that contains the word NOT. To negate an EXISTS predicate, precede it with the logical operator NOT, as follows:

```
NOT EXISTS (fullselect)
```

The result is then false if the EXISTS predicate is true, and true if the predicate is false. Here, NOT is a logical operator and not a part of the predicate. Logical operators are discussed in "Search conditions" on page 324.

Example 1: The following query lists the employee number of everyone represented in DSN8B10.EMP who works in a department where at least one employee has a salary less than 20000. Like many EXISTS predicates, the one in this query involves a correlated variable.

```
SELECT EMPNO
  FROM DSN8B10.EMP X
  WHERE EXISTS (SELECT * FROM DSN8B10.EMP
                  WHERE X.WORKDEPT=WORKDEPT AND SALARY<20000);
```

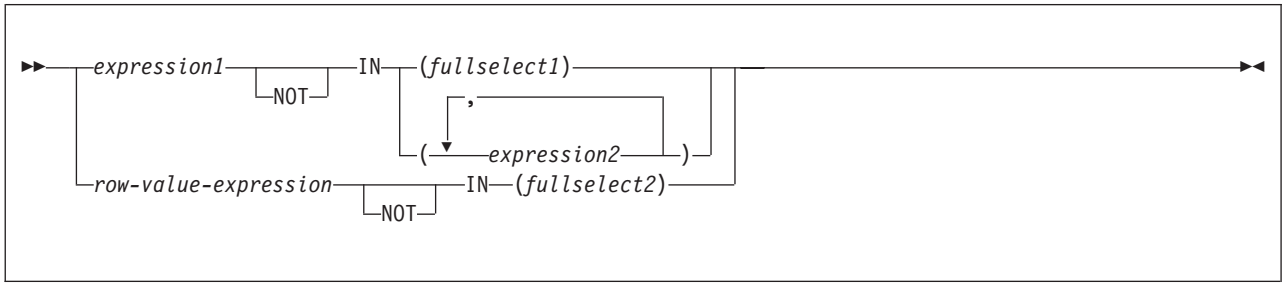
Example 2: List the subscribers (SNO) in the state of California who made at least one call during the first quarter of 2009. Order the results according to SNO. Each MONTH*nn* table has columns for SNO, CHARGES, and DATE. The CUST table has columns for SNO and STATE.

```
SELECT C.SNO
  FROM CUST C
  WHERE C.STATE = 'CA'
  AND EXISTS (
  SELECT *
    FROM MONTH1
    WHERE DATE BETWEEN '01/01/2009 AND '01/31/2009'
    AND C.SNO = SNO
  UNION ALL
  SELECT *
```

```
  FROM MONTH2
  WHERE DATE BETWEEN '02/01/2009 AND '02/28/2009'
  AND C.SNO = SNO
UNION ALL
SELECT *
FROM MONTH3
WHERE DATE BETWEEN '03/01/2009 AND '03/31/2009'
AND C.SNO = SNO
)
ORDER BY C.SNO;
```

IN predicate

The IN predicate compares a value or values with a set of values.



When *expression1* is specified, the IN predicate compares a value with a set of values. When *fullselect1* is specified, the fullselect must return a single result column, and can return any number of values, whether null or not null. The data type of *expression1* and the data type of the result column of *fullselect1* or *expression2* must be compatible. If *expression* is a single host variable, the host variable can identify a structure. Any host variable or structure that is specified must be described in the application program according to the rules for declaring host structures and variables.

When a *row-value-expression* is specified, the IN predicate compares values with a collection of values. The result table of the *fullselect2* must have the same number of columns as the *row-value-expression*. The data type of each expression in *row-value-expression* and the data type of its the corresponding result column of *fullselect2* must be compatible. The value of each expression in *row-value-expression* is compared with the value of its corresponding result column of *fullselect2*. The value of the predicate depends on the operator that is specified:

- When the operator is IN, the result of the predicate is:
  - True if at least one row returned from *fullselect2* is equal to the *row-value-expression*.
  - False if the result of *fullselect2* is empty or if no row returned from *fullselect2* is equal to the *row-value-expression*.
  - Otherwise, unknown (that is, if the comparison of *row-value-expression* to the row returned from *fullselect2* evaluates to unknown because of a null value for at least one row returned from *fullselect2* and no row returned from *fullselect2* is equal to the *row-value-expression*).
- When the operator is NOT IN, the result of the predicate is:

  True if the result of *fullselect2* is empty or if the *row-value-expression* is not equal to any of the rows returned by *fullselect2*.

  False if the *row-value-expression* is equal to at least one row returned by*fullselect2*.

  Otherwise, unknown (that is, if the comparison of *row-value-expression* to the row returned from *fullselect2* evaluates to unknown because of a null value for at least one row returned from *fullselect2* and the comparison of *row-value-expression* to the row returned from *fullselect2* is not true for any row returned by the *fullselect2*).

The IN predicate is equivalent to the quantified predicate as follows:

Table 52. IN predicate and equivalent quantified predicates

IN predicate	Equivalent quantified predicate
expression1 IN (*expression2*)	*expression1* = *expression2*
expression IN (*fullselect1*)	*expression* = ANY (*fullselect1*)
expression NOT IN (*fullselect1*)	*expression* <> ALL (*fullselect1*)
expression1 IN (*expressiona*, *expressionb*, ...)	*expression1* IN (SELECT * FROM R) When T is a table with a single row and R is a result table formed by the following fullselect: SELECT value1 FROM T UNION SELECT value2 FROM T UNION . . . UNION SELECT valuen FROM T
row-value-expression IN (*fullselect2*)	*row-value-expression* = SOME (*fullselect2*)
row-value-expression IN (*fullselect2*)	*row-value-expression* = ANY (*fullselect2*)
row-value-expression NOT IN (*fullselect2*)	*row-value-expression* <> ALL (*fullselect2*)

If the operands of the IN predicate have different data types or attributes, the rules that are used to determine the data type for evaluation of the IN predicate are those for UNION, EXCEPT, and INTERSECT. For a description, see "Rules for result data types" on page 144.

If the operands of the IN predicate are strings with different CCSIDs, the rules used to determine which operands are converted are those for operations that combine strings. See "Character and graphic string comparisons" on page 135.

Example 1: The following predicate is true for any row whose employee is in department D11, B01, or C01.

```
WORKDEPT IN ('D11', 'B01', 'C01')
```

Example 2: The following predicate is true for any row whose employee works in department E11.

```
EMPNO IN (SELECT EMPNO FROM DSN8B10.EMP
  WHERE WORKDEPT = 'E11')
```

Example 3: The following predicate is true if the date that a project is estimated to start (PRENDATE) is within the next two years.

```
YEAR(PRENDATE)  IN (YEAR(CURRENT DATE),
                    YEAR(CURRENT DATE + 1 YEAR),
                    YEAR(CURRENT DATE + 2 YEARS))
```

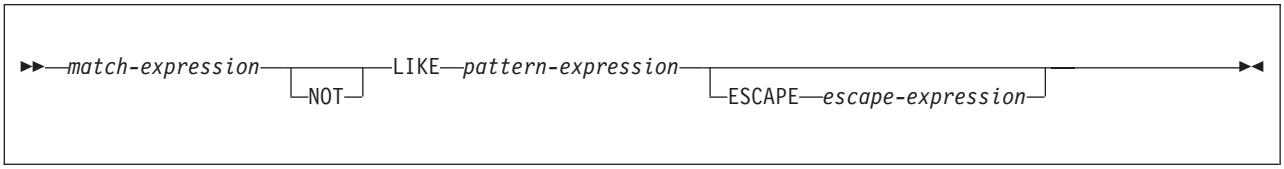
Example 4: The following example obtains the phone number of an employee in DSN8B10.EMP where the employee number (EMPNO) is a value specified within the COBOL structure defined below.

```
77 PHNUM  PIC X(6).
01 EMPNO-STRUCTURE.
   05 CHAR-ELEMENT-1  PIC X(6) VALUE '000140'.
   05 CHAR-ELEMENT-2  PIC X(6) VALUE '000340'.
```

```
   05 CHAR-ELEMENT-3  PIC X(6) VALUE '000220'.
   .
   .
   .
EXEC SQL DECLARE PHCURS CURSOR FOR
     SELECT PHONENO FROM DSN8B10.EMP
       WHERE EMPNO IN
       (:EMPNO-STRUCTURE.CHAR-ELEMENT-1,
        :EMPNO-STRUCTURE.CHAR-ELEMENT-2,
        :EMPNO-STRUCTURE.CHAR-ELEMENT-3)
END-EXEC.
EXEC SQL OPEN PHCURS
END-EXEC.
EXEC SQL FETCH PHCURS INTO :PHNUM
END-EXEC.
```

LIKE predicate

The LIKE predicate searches for strings that have a certain pattern.



The *match-expression* is the string to be tested for conformity to the pattern specified in *pattern-expression*. Underscore and percent sign characters in the pattern have a special meaning instead of their literal meanings unless *escape-expression* is specified. For more information, see the description of *pattern-expression*.

The following rules summarize how a predicate in the form of *m* `LIKE` *p* is evaluated:

- If *m* or *p* is null, the result of the predicate is unknown.
- If *m* and *p* are both empty, the result of the predicate is true.
- If *m* is empty and *p* is not, the result of the predicate is unknown unless *p* consists of one or more percent signs.
- If *m* is not empty and *p* is empty, the result of the predicate is false.
- Otherwise, if *m* matches the pattern in *p*, the result of the predicate is true. The description of *pattern-expression* provides a detailed explanation on how the pattern is matched to evaluate the predicate to true or false.

  The way the pattern is matched to evaluate the predicate changes when *LIKE blank insignificant* behavior is enabled. For more information, see LIKE blank insignificant behavior.

The values for *match-expression*, *pattern-expression*, and *escape-expression* must all be character or graphic strings or a mixture of both or they must all be binary strings (BLOBs). None of the expressions can yield a distinct type; however, an expression can be a function that casts a distinct type to its source type.

There are slight differences in what expressions are supported for each argument. The description of each argument lists the supported expressions.

match-expression
An expression that specifies the string to be tested for conformity to a certain pattern of characters.

LIKE *pattern-expression*
An expression that specifies the pattern of characters to be matched.

The expression can be specified by any one of the following:
- A constant
- A special register
- A variable
- A scalar function whose arguments are any of the above (though nested function invocations cannot be used)
- An array element specification
- A CAST specification whose arguments are any of the above
- An expression that concatenates (using CONCAT or `||`) any of the above

The expression must also meet these restrictions:

- The maximum length of *pattern-expression* must not be larger than 4000 bytes.
- If a host variable is used in *pattern-expression*, the host variable must be defined in accordance with the rules for declaring string host variables and must not be a structure.
- If *escape-expression* is specified, *pattern-expression* must not contain the escape character that is identified by *escape-expression*, except when immediately followed by the escape character, '%', or '_'. For example, if '+' is the escape character, any occurrences of '+' other than '++', '+_', or '+ in the pattern is an error.

When the pattern specified in a LIKE predicate is a parameter marker and a fixed-length character host variable is used to replace the parameter marker, specify a value for the host variable that is the correct length. If you do not specify the correct length, the select does not return the intended results. For example, if the host variable is defined as CHAR(10) and the value WYSE% is assigned to that host variable, the host variable is padded with blanks on assignment. The pattern used is 'WYSE ', which requests DB2 to search for all values that start with WYSE and end with five blank spaces. If you intended to search for only the values that start with 'WYSE ', you should assign the value 'WYSE%%%%%' to the host variable.

If the pattern is specified in a fixed-length string variable, any trailing blanks are interpreted as part of the pattern. Therefore, it is better to use a varying-length string variable with an actual length that is the same as the length of the pattern. If the host language does not allow varying-length string variables, place the pattern in a fixed-length string variable whose length is the length of the pattern.

For more information about the use of host variables with specific programming languages, see Host variables (DB2 Application programming and SQL).

The pattern is used to specify the conformance criteria for values in the *match-expression* where:

- The underscore character (_) represents any single character.
- The percent sign (%) represents a string of zero or more characters.
- Any other character represents a single occurrence of itself.

If the *pattern-expression* must include either the underscore or the percent character, the *escape-expression* is used to specify a character to precede either the underscore or percent character in the pattern. For character strings, the terms *character*, *percent sign*, and *underscore* refer to SBCS characters. For graphic strings, the terms refer to double-byte or UTF-16 characters.

A rigorous description of the pattern: This more rigorous description of the pattern ignores the use of the *escape-expression*.

Let *m* denote the value of *match-expression* and let *p* denote the value of *pattern-expression*. The string *p* is interpreted as a sequence of the minimum number of substring specifiers so each character of *p* is part of exactly one substring specifier. A substring specifier is an underscore, a percent sign, or any non-empty sequence of characters other than an underscore or a percent sign.

The result of the predicate is unknown if *m* or *p* is the null value. Otherwise, the result is either true or false. The result is true if *m* and *p* are both empty strings or there exists a partitioning of *m* into substrings such that:

- A substring of *m* is a sequence of zero or more contiguous characters and each character of *m* is part of exactly one substring.
- If the *n*th substring specifier is an underscore, the *n*th substring of *m* is any single character.
- If the *n*th substring specifier is a percent sign, the *n*th substring of *m* is any sequence of zero or more characters.
- If the *n*th substring specifier is neither an underscore nor a percent sign, the *n*th substring of *m* is equal to that substring specifier and has the same length as that substring specifier.
- The number of substrings of *m* is the same as the number of substring specifiers.

It follows that if *p* is an empty string and *m* is not an empty string, the result is false. Similarly, if *m* is an empty string and *p* is not an empty string consisting of a value other than percentage signs, the result is false.

The predicate *m* `NOT LIKE` *p* is equivalent to the search condition `NOT (`*m* `LIKE` *p*`)`.

Mixed data patterns: If *match-expression* represents mixed data, the pattern is assumed to be mixed data. For ASCII and EBCDIC, the special characters in the pattern are interpreted as follows:

- An SBCS underscore refers to one SBCS character.
- A DBCS underscore refers to one MBCS character.
- A percent sign (either SBCS or DBCS) refers to a string of zero or more SBCS or MBCS characters.

For EBCDIC, redundant shift bytes in *match-expression* or *pattern-expression* are ignored.

For Unicode, the special characters in the pattern are interpreted as follows:

- An SBCS or DBCS underscore refers to one character (either SBCS or MBCS).
- A percent sign (either SBCS or DBCS) refers to a string of zero or more SBCS or MBCS characters.

When the LIKE predicate is used with Unicode data, the Unicode percent sign and underscore use the code points indicated in the following table:

Character	UTF-8	UTF-16
Half-width %	X'25'	X'0025'
Full-width %	X'EFBC85'	X'FF05'
Half-width_	X'5F'	X'005F'
Full-width_	X'EFBCBF'	X'FF3F'

The full-width or half-width % matches zero or more characters. The full-width or half-width_ character matches exactly one character. (For ASCII or EBCDIC

data, a full-width _ character matches one DBCS character.)

ESCAPE *escape-expression*

An expression that specifies the escape character to be used to modify the special meaning of the underscore (_) and percent (%) characters in *pattern-expression*. Specifying an expression, which is optional, allows the LIKE predicate to explicitly test that the value contains a '%' or '_' in the character positions that you want. The escape character consists of a single SBCS (1 byte) or DBCS (2 bytes) character. An escape clause is allowed for Unicode mixed (UTF-8) data, but is restricted for ASCII and EBCDIC mixed data.

The expression can be specified by:

- A constant
- A variable
- A scalar function whose arguments are any of the above (though nested function invocations cannot be used)
- A CAST specification whose arguments are any of the above

The following rules also apply to the use of the **ESCAPE** clause and *escape-expression*:

- The result of *escape-expression* must be one SBCS or DBCS character or a binary string that contains exactly 1 byte.
- The **ESCAPE** clause cannot be used if *match-expression* is mixed data.
- If *escape-expression* is specified by a host variable, the host variable must be defined in accordance with the rules for declaring fixed-length string host variables.[16] If the host variable has a negative indicator variable, the result of the predicate is unknown.
- The pattern must not contain the escape character except when followed by the escape character, '%' or '_'. For example, if '+' is the escape character, any occurrences of '+' other than '++', '+_', or '+%' in the pattern is an error.

The following table shows the effect of successive occurrences of the escape character, which in this case is the plus sign (+).

Table 53. Effect of successive occurrences of the escape character

When the pattern string is...	The actual pattern is...
+%	A percent sign
++%	A plus sign followed by zero or more arbitrary characters
+++%	A plus sign followed by a percent sign

Examples

Example 1: The following predicate is true when the string to be tested in NAME has the value SMITH, NESMITH, SMITHSON, or NESMITHY. It is not true when the string has the value SMYTHE:

```
NAME LIKE '%SMITH%'
```

Example 2: In the predicate below, a host variable named PATTERN holds the string for the pattern:

```
NAME LIKE :PATTERN ESCAPE '+'
```

16. If it is NUL-terminated, a C character string variable of length 2 can be specified.

Assume that the string in PATTERN has the following value:

```
AB+_C_%
```

Observe that in this string, the plus sign preceding the first underscore is an escape character. The predicate is true when the string being tested in NAME has the value AB_CD or AB_CDE. It is false when this string has the value AB, AB_, or AB_C.

Example 3: The following two predicates are equivalent; three of the four percent signs in the first predicate are redundant.

```
NAME LIKE 'AB%%%CD'
NAME LIKE 'AB%CD'
```

Example 4: Assume that a distinct type named ZIP_TYPE with a source data type of CHAR(5) exists and an ADDRZIP column with data type ZIP_TYPE exists in some table TABLEY. The following statement selects the row if the zip code (ADDRZIP) begins with '9555'.

```
SELECT * FROM TABLEY
  WHERE CHAR(ADDRZIP) LIKE '9555%'
```

Example 5: The RESUME column in sample table DSN8B10.EMP_PHOTO_RESUME is defined as a CLOB. The following statement selects the RESUME column when the string JONES appears anywhere in the column.

```
SELECT RESUME FROM DSN8B10.EMP_PHOTO_RESUME
  WHERE RESUME LIKE '%JONES%'
```

Example 6: In the following table, assume COL1 is a column that contains mixed EBCDIC data. The table shows the results when the predicate in the first column is evaluated using the COL1 value in the second column:

Predicates	COL1 Values	Result
WHERE COL1 LIKE 'aaa<SO>AB%C<SI>'	'aaa<SO>ABDZC<SI>'	True
	'aaa<SO>AB<SI>dzx<SO>C<SI>'	True
WHERE COL1 LIKE 'a%<SO>C<SI>'	'a<SO>C<SI>'	True
	'ax<SO>C<SI>'	True
	'ab<SO>DE<SI>fg<SO>C<SI>'	True
WHERE COL1 LIKE 'a_<SO>C<SI>'	'a%<SO>C<SI>'	True
	'a<SO>XC<SI>'	False
WHERE COL1 LIKE 'a<SO>__C<SI>'	'a<SO>XC<SI>'	True
	'ax<SO>C<SI>'	False
WHERE COL1 LIKE '<SO><SI>'	Empty string	True
WHERE COL1 LIKE 'ab<SO>C<SI>_'	'ab<SO>C<SI>d'	True
	'ab<SO><SI><SO>C<SI>d'	True

Example 7: In the following table, assume COL1 is a column that contains mixed ASCII data. The table shows the results when the predicate in the first column is evaluated using the COL1 value in the second column:

Predicates	COL1 Values	Result
WHERE COL1 LIKE 'aaa𝐀𝐁%𝐂'	'aaa𝐀𝐁𝐃𝐙𝐂'	True
WHERE COL1 LIKE 'aaa𝐀𝐁%𝐂'	'aaa𝐀𝐁dzx𝐂'	True

Example 8: In the following table, assume COL1 is a column that contains Unicode data. The table shows the results when the predicate in the first column is evaluated using the COL1 value in the second column:

Table 54. COL1 contain Unicode data

Predicates	COL1 values	Result
WHERE COL1 LIKE 'aaaAB%C	'aaaABDZC'	True
	'aaaABdzxC'	True
	empty string	False
WHERE COL1 LIKE 'aaaAB %C'	'aaaABDZC'	True
	'aaaABdzxC'	True
	empty string	False
WHERE COL1 LIKE ''	'aaaABDZC'	False
	'aaaABdzxC'	False
	empty string	True
WHERE COL1 LIKE '%'	'aaaABDZC'	True
	'aaaABdzxC'	True
	empty string	True
WHERE COL1 LIKE ' %'	'aaaABDZC'	True
	'aaaABdzxC'	True
	empty string	False
WHERE COL1 LIKE ' '	'aaaABDZC'	False
	'aaaABdzxC'	False
	empty string	False

LIKE blank insignificant behavior

When the LIKE_BLANK_INSIGNIFICANT subsystem parameter is enabled, all of the blanks at the end of a fixed-length string are ignored. This behavior is called *LIKE blank insignificant* behavior. *LIKE blank significant* behavior, in which the blanks at the end of fixed-length strings are significant (not ignored), is the default behavior during installation or migration. For variable length strings, blanks are significant.

When you set the LIKE_BLANK_INSIGNIFICANT subsystem parameter, LIKE blank insignificant behavior takes effect the next time an SQL query statement with the LIKE predicate is executed after the statement is bound or prepared. If the statement is not prepared or bound, the LIKE behavior exhibits LIKE blank significant behavior regardless of the subsystem parameter setting.

For the following functions, enabling or disabling LIKE blank insignificant behavior takes effect immediately. This applies to both an explicit LIKE predicate (for example, UNLOAD) and an implicit LIKE predicate (for example, table check constraint).

- INSERT
- UPDATE
- UNLOAD
- REORG
- LOAD
- CHECK DATA

Before the LIKE predicate is applied, any trailing blanks in a CHARACTER or GRAPHIC column are stripped to the last non-blank character. If the column contains all blanks, the blank in character position 1 is not stripped. After stripping occurs, the LIKE predicate is applied against the stripped column data.

Tip: After you enable the LIKE_BLANK_INSIGNIFICANT subsystem parameter, existing rows might not conform to table check constraints that contain a LIKE predicate. Consider running the CHECK DATA utility on all affected tables to find the records that do not conform to the table check constraint.

The following examples, in which *b* represents a blank character, demonstrate how the LIKE predicate is evaluated when LIKE blank insignificant behavior is enabled.

```
SELECT C1
FROM T1
WHERE C1 LIKE '%xyz';
```

This LIKE predicate will match the following fixed-length strings:

- abcxyz
- abcxyz*b*
- abcxyz*bb*
- abcxyz*bb..b*'

While trailing blanks in the column data are insignificant, trailing blanks in the LIKE predicate are significant. The following example, in which*b* represents a blank character, applies to when the LIKE predicate contains one or more trailing blanks.

```
SELECT C1
FROM T1
WHERE C1 LIKE '%xyzbb';
```

This LIKE predicate will not match the following fixed-length strings:

- abcxyz
- abcxyz*b*
- abcxyz*bb*
- abcxyz*bbb*

The following example applies to when the LIKE predicate contains one or more single characters (_) in the last position.

```
SELECT C1
FROM T1
WHERE C1 LIKE '%xyz_';
```

This LIKE predicate will not match the following fixed-length strings, because they are all stripped to the 'abcxyz' string:

- abcxyz
- abcxyz*b*
- abcxyz*bb*
- abcxyz*bb..b*

The following example applies to when the LIKE predicate contains more than one single character (_) in the last position.

```
SELECT C1
FROM T1
WHERE C1 LIKE '%xyz__';
```

This LIKE predicate will not match the following fixed-length strings:

- abcxyz
- abcxyz*b*
- abcxyz*bb*
- abcxyz*bb..b*

The following example applies to when the LIKE predicate contains more than one single character (_) and a string of zero or more characters (%) are in the last positions.

```
SELECT C1
FROM T1
WHERE C1 LIKE '%xyz_%_';
This
```

This LIKE predicate will not match the following fixed-length strings:

- abcxyz
- abcxyz*b*
- abcxyz*bb*
- abcxyz*bb..b*

If the column data contains all blanks, every blank, except the blank in character position one, is stripped before the LIKE predicate is applied. For example, a CHAR(6) column contains the following values:

bbbbbb

The following LIKE predicates will match:

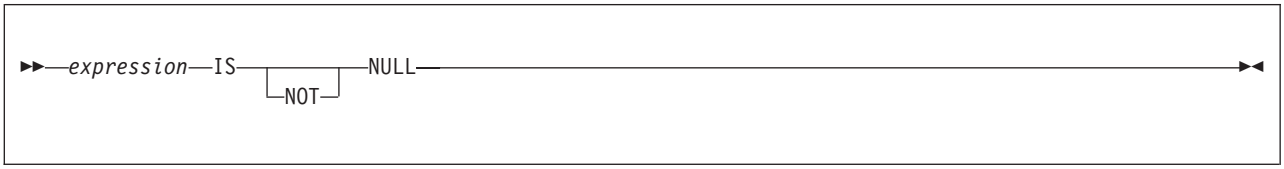
- LIKE '*b*'
- LIKE '_'
- LIKE '%'

The following LIKE predicates will not match:

- LIKE '*bbbbbb*'
- LIKE '*bbbb*___'
- LIKE '_______'

NULL predicate

The NULL predicate tests for null values.



The result of a NULL predicate cannot be unknown. If the value of the expression is null, the result is true. If the value is not null, the result is false. If NOT is specified, the result is reversed.

A parameter marker must not be specified for or within the expression.

Example 1: The following predicate is true whenever PHONENO has the null value, and is false otherwise.

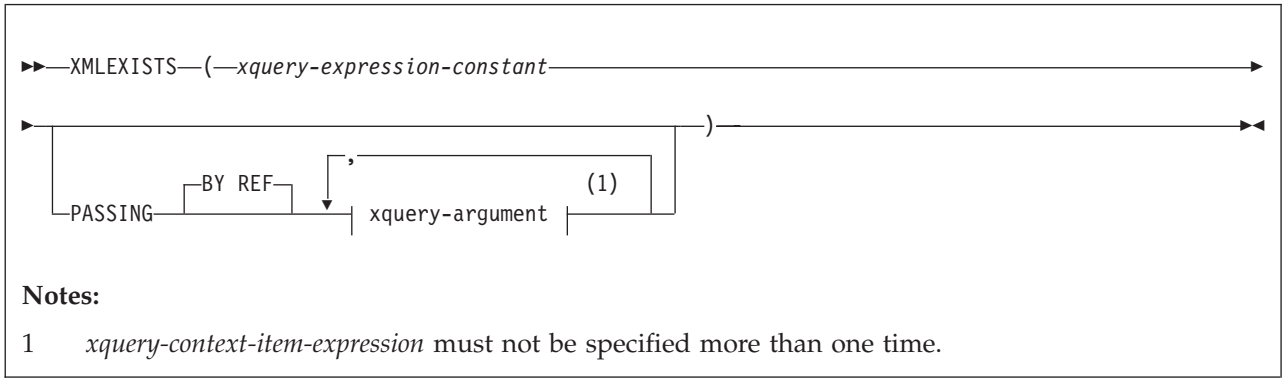
```
PHONENO IS NULL
```

Example 2: The following predicate is true whenever the array MYARRAY has the null value, and is false otherwise.

```
MYARRAY IS NULL
```

XMLEXISTS predicate

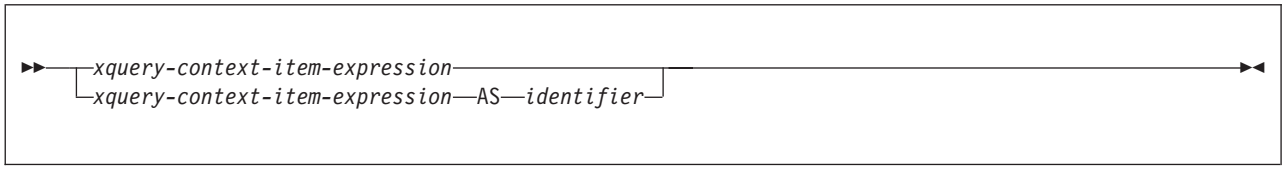
The XMLEXISTS predicate tests whether an XQuery expression returns a sequence of one or more items.



Notes:

1 *xquery-context-item-expression* must not be specified more than one time.

xquery-argument



xquery-expression-constant
Specifies a character string constant that is interpreted as an XQuery expression using supported XQuery language syntax. See *DB2 XML Guide* for information about the XQuery language syntax. *xquery-expression-constant* cannot be an XQuery updating expression. The XQuery expression is evaluated with the arguments specified in *xquery-argument*. *xquery-expression-constant* must not be an empty string or a string of all blanks.

PASSING
Specifies input values and the manner in which these values are passed to the XQuery expression specified by *xquery-expression-constant*.

BY REF
Specifies that the XML input value arguments are to be passed by reference. When XML values are passed by reference, the XQuery evaluation uses the input node trees, preserving all properties including the original node identities and document order. If two arguments pass the same XML value, node identity comparisons and document ordering comparisons that involve some nodes that are contained between the two input arguments might refer to nodes within the same XML node tree.

This clause has no impact on how non-XML values are passed. The non-XML values create a new copy of the value during the cast to XML.

xquery-argument
Specifies an argument to use in the evaluation of the XQuery expression specified by *xquery-expression-constant*. A query argument is an expression that returns a value that is XML, integer, decimal, or a character or graphic string that is not a LOB. *xquery-argument* must not return ROWID, TIMESTAMP,

binary string, REAL, DECFLOAT data types, or a character string data type that is bit data, and must not reference a sequence expression or a *OLAP-specification*.

An argument specifies a value and the manner in which that value is to be passed. How an argument in the PASSING clause is used in the XQuery expression depends on whether the argument is specified as the *xquery-context-item-expression* or an *xquery-variable-expression*. The argument includes an SQL expression that is evaluated before passing the result to the XQuery expression.

- If the resulting value is an XML value, it becomes an *input-xml-value*. It is passed by reference which means that the original values, not copies, are used in the evaluation of the XQuery expression.
- If the resulting value is not an XML value, the result of the expression must be able to be cast to an XML value. The cast value becomes an *input-xml-value*. An empty string is converted to an XML empty string.
- If the resulting value is a null value, it is converted to an XML empty sequence if the argument is *xquery-variable-expression*. If the argument is *xquery-context-expression*, the XMLEXISTS predicates returns unknown.

xquery-context-item-expression

xquery-context-item-expression specifies the *initial context item* in the XQuery expression specified by *xquery-expression-constant*. The value of the initial context item is the result of *xquery-context-item-expression* cast to XML. *xquery-context-item-expression* must not be specified more than one time.

xquery-context-item-expression must not be a sequence of more than one item. If the result of *xquery-context-item-expression* is an empty string, the XQuery expression is evaluated with the initial context item set to an XML empty string.

If the *xquery-context-item-expression* is not specified or is an empty string, the initial context item in the XQuery expression is undefined, and the XQuery expression must not reference the initial context item. An XQuery variable is not created for the context item expression.

If the *xquery-context-expression* is not specified or the *input-xml-value* that results from the *xquery-context-expression* is an XML empty sequence, the initial context item is undefined. If the XQuery expression refers to the initial context item, it must be specified with a value that is not an XML empty sequence.

xquery-variable-expression

xquery-variable-expression specifies an argument to the XQuery expression. An XQuery variable is created for each *xquery-variable-expression*, and the XQuery variable is set to the result of *xquery-argument-expression* cast to XML. If the result of *xquery-variable-expression* is an empty string, the XQuery variable is set to an XML empty string. If *xquery-variable-expression* is null, the XQuery variable is set to an XML empty sequence. For example, `PASSING T.A + T.B as "sum"` creates an XQuery variable named sum. The scope of the XQuery variables created from the PASSING clause is the XQuery expression that is specified by *xquery-expression-constant*.

AS *identifier*

Specifies that the value that is generated by *xquery-variable-expression* will be passed to *xquery-expression-constant* as an XQuery variable named *identifier*. The length of the name must not be longer than 128 bytes. The leading dollar sign ($) that precedes variable names in the XQuery language is not included in

identifier. The name must be an XML NCName that is not the same as *identifier* for another *xquery-variable-expression* in the same PASSING clause.

The result of the predicate is determined as follows:

- The result is unknown if *xquery-context-item-expression* specified in the PASSING clause is a NULL value
- the result is false if the result of the XQuery expression is an empty sequence
- the result is true in all other cases

If the evaluation of the XQuery expression results in an error, XMLEXISTS returns an error. The XMLEXISTS predicate is not supported in ON clause of outer joins.

Example: Find all the purchase orders that buy a baby monitor. This example finds the product number for baby monitors from the product table and joins the result to the PurchaseOrders table. It then evaluates the XQuery expression //item[@partnum = *$n*] for each row and returns those rows that contain an item element node with a partNum attribute that is equal to the product number of 'Baby Monitor'. The context item for the XQuery expression is PO.POrder. An XQuery variable, *$n*, is created and initialized to the value of S.prodno:

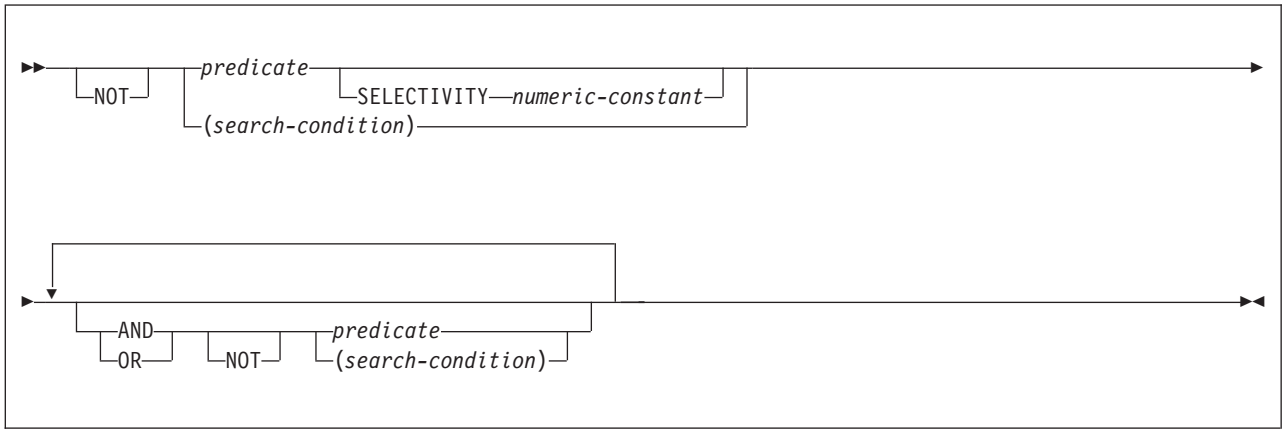
```
SELECT S.prodno, count(*) as result
   FROM PurchaseOrders PO, Products S
   WHERE XMLEXISTS ('//item[@partNum = $n]'
                    PASSING PO.POrder,
                            S.prodno AS "n")
   AND S.prod_name = 'Baby Monitor';
```

The results might be similar to the following:

```
Prodno      result
------------------------------

926-AA      1
```

Search conditions

A *search condition* specifies a condition that is true, false, or unknown about a given row or group. When the condition is true, the row or group qualifies for the results. When the condition is false or unknown, the row or group does not qualify.



Description

SELECTIVITY *numeric-constant*

Specifies the expected selectivity percentage for the predicate. You can specify the SELECTIVITY clause only when the predicate contains one of the indexable spatial predicate functions and the predicate is in the form of *spatial-predicate-function operator expression*, where *operator* is either = or <. The selectivity value must be an integer or decimal constant value in the range from 0 to 1 (inclusive). For example, if you specify 0.01, the spatial predicate function is expected to filter out all but one percent of all the rows in the table. An error is returned if the SELECTIVITY clause is specified for a non-spatial predicate function.

The result of a search condition is derived by application of the specified *logical operators* (AND, OR, NOT) to the result of each specified predicate. If logical operators are not specified, the result of the search condition is the result of the specified predicate.

AND and OR are defined in the following table, in which *P* and *Q* are any predicates:

Table 55. Truth table for AND and OR

P	*Q*	*P* **and** *Q*	*P* **or** *Q*
True	True	True	True
True	False	False	True
True	Unknown	Unknown	True
False	True	False	True
False	False	False	False
False	Unknown	False	Unknown
Unknown	True	Unknown	True

Table 55. Truth table for AND and OR (continued)

P	*Q*	*P* **and** *Q*	*P* **or** *Q*
Unknown	False	False	Unknown
Unknown	Unknown	Unknown	Unknown

NOT(true) is false and NOT(false) is true, but NOT(unknown) is still unknown. The NOT logical operator has no affect on an unknown condition. The result of NOT(unknown) is still unknown.

Search conditions within parentheses are evaluated first. If the order of evaluation is not specified by parentheses, NOT is applied before AND, and AND is applied before OR. The order in which operators at the same precedence level are evaluated is undefined to allow for optimization of search conditions.

Example 1: In the first of the search conditions below, AND is applied before OR. In the second, OR is applied before AND.

```
SALARY>:SS AND  COMM>:CC OR BONUS>:BB
SALARY>:SS AND (COMM>:CC OR BONUS>:BB)
```

Example 2: In the first of the search conditions below, NOT is applied before AND. In the second, AND is applied before NOT.

```
NOT SALARY>:SS  AND  COMM>:CC
NOT (SALARY>:SS AND  COMM>:CC)
```

Example 3: For the following search condition, AND is applied first. After the application of AND, the OR operators could be applied in either order without changing the result. DB2 can therefore select the order of applying the OR operators.

```
SALARY>:SS AND COMM>:CC OR BONUS>:BB OR SEX=:GG
```

Options affecting SQL

Certain DB2 precompiler or coprocessor options (referred to as SQL processing options), DB2 subsystem parameters (set through the installation panels), bind options, options for CREATE PROCEDURE and ALTER PROCEDURE statements for native SQL procedures, and special registers affect how SQL statements can be composed or determine how SQL statements are processed.

The following table summarizes the effect of these options and shows where to find more information. (Some of the items are described in detail following the table, while other items are described elsewhere.)

Table 56. Summary of items affecting composition and processing of SQL statements

SQL processing option	Other[1]	Affects
	DYNAMICRULES bind option or the native SQL procedures option	The rules that DB2 applies to dynamic SQL statements. For details about authorization, see "Authorization IDs and dynamic SQL" on page 75. The option can also affect decimal point representation, string delimiters, and decimal arithmetic. For details about how DB2 applies the options to dynamic SQL statements when DYNAMICRULES bind, define, or invoke behavior is in effect, see "SQL processing options for dynamic statements" on page 327.
	USE FOR DYNAMICRULES	Use of options for dynamic statements when DYNAMICRULES bind, define, or, invoke behavior is in effect. For details, see "SQL processing options for dynamic statements" on page 327.
COMMA PERIOD	DECIMAL POINT IS	Representation of decimal points in SQL statements. For details, see "Decimal point representation" on page 328.
APOSTSQL QUOTESQL	SQL STRING DELIMITER	Representation of string delimiters in SQL statements. For details, see "Apostrophes and quotation marks as string delimiters" on page 330.
	ASCII CCSID	A numeric value that determines the CCSID of ASCII string data. For details, see "Mixed data in character strings" on page 331.
	EBCDIC CCSID	A numeric value that determines the CCSID of EBCDIC string data and whether Katakana characters can be used in ordinary identifiers. For details, see "Katakana characters for EBCDIC" on page 331.
	UNICODE CCSID	A numeric value that determines the CCSID of Unicode string data. For details, see "Mixed data in character strings" on page 331.
CCSID	MIXED DATA	Use of ASCII or EBCDIC character strings with a mixture of SBCS and DBCS characters. For details, see "Mixed data in character strings" on page 331.
DATE TIME	DATE FORMAT TIME FORMAT LOCAL DATE LENGTH LOCAL TIME LENGTH	Formatting of datetime strings. For details, see "Formatting of datetime strings" on page 332.
STDSQL		Conformance with the SQL standard. For details, see "SQL standard language" on page 332.

Table 56. Summary of items affecting composition and processing of SQL statements (continued)

SQL processing option	Other[1]	Affects
NOFOR or STDSQL		Whether the FOR UPDATE clause must be specified (in the SELECT statement of the DECLARE CURSOR statement). For details, see "Positioned updates of columns" on page 333.
CONNECT		Whether the rules for the CONNECT(1) or CONNECT(2) SQL processing option apply. For details about the SQL processing option, see *DB2 Application Programming and SQL Guide*.
	SQLRULES bind option	Whether a CONNECT statement is processed with DB2 rules or SQL standard rules.
	CURRENT RULES special register	Whether the statements ALTER TABLE, CREATE TABLE, GRANT, and REVOKE are processed with DB2 rules or SQL standard rules. For details, see "CURRENT RULES" on page 189. Whether DB2 automatically creates the LOB table space, auxiliary table, and index on the auxiliary table for a LOB column in a base table. For details, see Creating a table with LOB columns. Whether DB2 automatically creates an index on a ROWID column that is defined with GENERATED BY DEFAULT. For details, see the description of the clause for "CREATE TABLE" on page 1388. Whether an external stored procedure runs as a main or subprogram. For details, see "CREATE PROCEDURE (external)" on page 1319.
	SQLRULES bind option or CURRENT RULES special register	Whether SQLCODE +236 is issued when the SQLDA provided on DESCRIBE or PREPARE INTO is too small and the result columns do not involve LOBs or distinct types. For details, see "DESCRIBE" on page 1590 and "SQL descriptor area (SQLDA)" on page 2079.
DEC	DECIMAL ARITHMETIC or CURRENT PRECISION special register	Whether DEC15 or DEC31 rules are used when both operands in a decimal operation have 15 digits or less. For details, see "Arithmetic with two decimal operands" on page 244.

Note: [1] The entries in this column are fields on installation panels unless otherwise noted.

For further details on SQL processing options, see *DB2 Application Programming and SQL Guide*. For more details on bind options, see *DB2 Command Reference*.

SQL processing options for dynamic statements

Generally, dynamic statements use the application programming defaults specified on installation panel DSNTIPF. However, if the value of installation panel field USE FOR DYNAMICRULES is NO and DYNAMICRULES bind, define, or invoke behavior is in effect, certain SQL processing options are used instead of the application programming defaults.

The following SQL processing options are used instead of the application programming defaults:
- COMMA or PERIOD
- APOST or QUOTE
- APOSTSQL or QUOTESQL
- DEC(15) or DEC(31)

For some languages, the SQL processing option defaults to a value and no alternative is allowed. If the value of installation panel field USE FOR DYNAMICRULES is YES, dynamic statements use the application programming defaults regardless of the value of DYNAMICRULES option.

For additional information on the effect of SQL processing options and application programming defaults on:
- Decimal point representation, see "Decimal point representation."
- String delimiters, see "Apostrophes and quotation marks as string delimiters" on page 330.
- Decimal arithmetic, see "Arithmetic with two decimal operands" on page 244.

For a list of the DYNAMICRULES option values that specify run, bind, define, or invoke behavior, see Table 6 on page 75.

DECFLOAT rounding mode

All views and SQL functions referenced in an SQL statement must either not have rounding mode information stored in the SYSENVIRONMENT catalog, or they must all have the same rounding mode information in the SYSENVIRONMENT catalog.

Decimal point representation

Decimal points in SQL statements are represented with either periods or commas.

Two values control the representation:
- The value of field DECIMAL POINT IS on installation panel DSNTIPF, which can be a comma (,) or period (.)
- COMMA or PERIOD, which are mutually exclusive SQL processing options for COBOL

These values apply to SQL statements as follows:
- For a distributed operation, the decimal point is the first of the following values that applies:
  - The decimal point value specified by the requester
  - The value of field DECIMAL POINT IS on panel DSNTIPF at the DB2 where the package is bound
- Otherwise:
  - For static SQL statements:
    - In a COBOL program, the SQL processing option COMMA or PERIOD determines the decimal point representation for every static SQL statement. If neither SQL processing option is specified, the value of DECIMAL POINT IS at precompilation time determines the representation.
    - In non-COBOL programs, the decimal representation for static SQL statements is always the period.
  - For dynamic SQL statements:

- If DYNAMICRULES run behavior applies, the decimal point is the value of field DECIMAL POINT IS on installation panel DSNTIPF at the local DB2 when the statement is prepared.

  For a list of the DYNAMICRULES option values that specify run, bind, define, or invoke behavior, see Table 6 on page 75.
- If DYNAMICRULES bind, define, or invoke behavior applies, and the value of installation panel field USE FOR DYNAMICRULES is YES, the decimal point is the value of field DECIMAL POINT IS.

  If bind, define, or invoke behavior applies, and field USE FOR DYNAMIC RULES is NO, the SQL processing option determines the decimal point representation. For COBOL programs, which supports SQL processing option COMMA or PERIOD, the decimal point representation is determined as described above for static SQL statements in COBOL programs. For programs written in other host languages, the default SQL processing option, which can only be PERIOD, is used.

If the comma is the decimal point, these rules apply:

- In any context, a comma intended as a separator must be followed by a space. Such commas could appear, for example, in a VALUES clause, an IN predicate, or an ORDER BY clause in which numbers are used to identify columns.
- In any context, a comma intended as a decimal point must not be followed by a space.
- If the DECIMAL POINT IS field (and not the SQL processing option) determines the comma as the decimal point, DB2 will recognize either a comma or a period as the decimal point in numbers in dynamic SQL.

Apostrophes and quotation marks as string delimiters

SQL processing options and DB2 installation panel fields control the representation of string delimiters in COBOL and SQL statements.

The following SQL processing options control the representation of string delimiters:

- APOST and QUOTE are mutually exclusive SQL processing options for COBOL. Their meanings are exactly what they are for the COBOL compilers:
  - APOST names the apostrophe (') as the string delimiter in COBOL statements.
  - QUOTE names the quotation mark (") as the string delimiter.

  Neither option applies to SQL syntax. Do not confuse them with the APOSTSQL and QUOTESQL options.
- APOSTSQL and QUOTESQL are mutually exclusive SQL processing options for COBOL. Their meanings are:
  - APOSTSQL names the apostrophe (') as the string delimiter and the quotation mark (") as the escape character in SQL statements.
  - QUOTESQL names the quotation mark (") as the string delimiter and the apostrophe (') as the escape character in SQL statements.

These values apply to SQL statements as follows:

- For a distributed operation, the string delimiter is the first of the following values that applies:
  - The SQL string delimiter value specified by the requester
  - The value of the field SQL STRING DELIMITER on installation panel DSNTIPF at the DB2 where the package is bound
- Otherwise:
  - For static SQL statements:

    In a COBOL program, the SQL processing option APOSTSQL or QUOTESQL determines the string delimiter and escape character. If neither SQL processing option is specified, the value of field SQL STRING DELIMITER on installation panel DSNTIPF determines the string delimiter and escape character.

    In a non-COBOL program, the string delimiter is the apostrophe, and the escape character is the quotation mark.
  - For dynamic SQL statements:
    - If DYNAMICRULES run behavior applies, the string delimiter and escape character is the value of field SQL STRING DELIMITER on installation panel DSNTIPF at the local DB2 when the statement is prepared.

      For a list of the DYNAMICRULES option values that specify run, bind, define, or invoke behavior, see Table 6 on page 75.
    - If DYNAMICRULES bind, define, or invoke behavior applies and the value of installation panel field USE FOR DYNAMICRULES is YES, the string delimiter and escape character is the value of field SQL STRING DELIMITER.

      If bind, define, or invoke behavior applies and USE FOR DYNAMICRULES is NO, the SQL processing option determines the string delimiter and escape character. For COBOL programs, SQL processing option APOSTSQL or QUOTESQL determines the string delimiter and escape character. If neither SQL processing option is specified, the value of field SQL STRING DELIMITER determines them. For programs written in other host

languages, the default SQL processing option, which can only be APOSTSQL, determines the string delimiter and escape character.

Katakana characters for EBCDIC

Ordinary identifiers with an EBCDIC encoding scheme can contain Katakana characters if the DB2 installation is set to allow it.

The field EBCDIC CCSID on installation panel DSNTIPF determines the system CCSIDs for EBCDIC-encoded data. Ordinary identifiers with an EBCDIC encoding scheme can contain Katakana characters if the field contains the value 5026 or 930. There are no corresponding SQL processing options. EBCDIC CCSID applies equally to static and dynamic statements. For dynamically prepared statements, the applicable value is always the one at the local DB2.

Mixed data in character strings

Mixed character data and graphic data are always allowed for Unicode, but for EBCDIC and ASCII, the specific installation of DB2 determines whether mixed data can be used.

The field MIXED DATA on installation panel DSNTIPF can have the value YES or NO for ASCII or EBCDIC character strings. The value YES indicates that character strings can contain a mixture of SBCS and DBCS characters. The value NO indicates that they cannot. Mixed character data and graphic data are always allowed for Unicode; that is the MIXED DATA field does not have an effect on Unicode data.

For static SQL statements, the value of the CCSID SQL processing option or the derived CCSID for the DB2 coprocessor determines whether ASCII or EBCDIC character strings can contain mixed data. If a mixed CCSID is used, mixed strings are allowed. If a single-byte CCSID is used, mixed strings are not allowed.

For dynamic SQL statements, the CCSID that is selected to convert the dynamic statement text to UTF-8 determines whether ASCII or EBCDIC character strings can contain mixed data. The CCSID for a dynamic statement is determined from the SQLDA override (if any) for the host variable on the PREPARE statement, the value of the CURRENT ENCODING SCHEME special register, and the ENCODING bind option.

The value of MIXED DATA affects the parsing of SQL character string constants, the execution of the LIKE predicate, and the assignment of character strings to host variables when truncation is needed. It can also affect concatenation, as explained in “Expressions with the concatenation operator” on page 250. A value that applies to a statement executed at the local DB2 also applies to any statement executed at another server. An exception is the LIKE predicate, for which the applicable value of MIXED DATA is always the one at the statement's server.

The value of MIXED DATA also affects the choice of system CCSIDs for the local DB2 and the choice of data subtypes for character columns. When this value is YES, multiple CCSIDs are available for ASCII and EBCDIC data (SBCS, DBCS, and MIXED). The CCSID specified in the ASCII CCSID or EBCDIC CCSID field is the MIXED CCSID. In this case, DB2 derives the SBCS and MIXED CCSIDs from the DBCS CCSID specified installation panel DSNTIPF. Moreover, a character column can have any one of the allowable data subtypes—BIT, SBCS, or MIXED.

On the other hand, when MIXED DATA is NO, the only ASCII or EBCDIC system CCSIDs are those for SBCS data. Therefore, only BIT and SBCS can be data subtypes for character columns.

Formatting of datetime strings

The format for a datetime string that is in effect for a statement that is executed at the local DB2 is not necessarily in effect for a statement that is executed at a different server.

Fields on installation panel DSNTIP4 (DATE FORMAT, TIME FORMAT, LOCAL DATE LENGTH, and LOCAL TIME LENGTH) and SQL processing options affect the formatting of datetime strings.

The formatting of datetime strings is described in "String representations of datetime values" on page 101. Unlike the subsystem parameters and options previously described, a value in effect for a statement executed at the local DB2 is not necessarily in effect for a statement executed at a different server. See "Restrictions on the use of local datetime formats" on page 105 for more information.

SQL standard language

DB2 SQL and the SQL standard are not identical. The STDSQL SQL processing option addresses some of the differences.

- STDSQL(NO) indicates that conformance with the SQL standard is not intended. The default is the value of field STD SQL LANGUAGE on installation panel DSNTIP4 (which has a default of NO).
- STDSQL(YES)[17] indicates that conformance with the SQL standard is intended.

When a program is precompiled with the STDSQL(YES) option, the following rules apply:

Declaring host variables: All host variable declarations except in Java and REXX must lie between pairs of BEGIN DECLARE SECTION and END DECLARE SECTION statements:

```
BEGIN DECLARE SECTION
 -- one or more host variable declarations
END DECLARE SECTION
```

Separate pairs of these statements can bracket separate sets of host variable declarations.

Declarations for SQLCODE and SQLSTATE: The programmer must declare host variables for either SQLCODE or SQLSTATE, or both. SQLCODE should be defined as a fullword integer and SQLSTATE should be defined as a 5-byte character string. SQLCODE and SQLSTATE cannot be part of any structure. The variables must be declared in the DECLARE SECTION of a program; however, SQLCODE can be declared outside of the DECLARE SECTION when no host variable is defined for SQLSTATE. For PL/I, an acceptable declaration can look like this:

```
DECLARE SQLCODE BIN FIXED(31);
DECLARE SQLSTATE CHAR(5);
```

17. STDSQL(86) is a synonym, but STDSQL(YES) should be used.

In Fortran programs, the variable SQLCOD should be used for SQLCODE, and either SQLSTATE or SQLSTA can be used for SQLSTATE.

Definitions for the SQLCA: An SQLCA must not be defined in your program, either by coding its definition manually or by using the INCLUDE SQLCA statement. When STDSQL(YES) is specified, the DB2 precompiler or coprocessor automatically generates an SQLCA that includes the variable name SQLCADE instead of SQLCODE and SQLSTAT instead of SQLSTATE. After each SQL statement executes, DB2 assigns status information to SQLCODE and SQLSTATE, whose declarations are described above, as follows:

- SQLCODE: DB2 assigns the value in SQLCADE to SQLCODE. In Fortran, SQLCAD and SQLCOD are used for SQLCADE and SQLCODE, respectively.
- SQLSTATE: DB2 assigns the value in SQLSTAT to SQLSTATE. (In Fortran, SQLSTT and SQLSTA are used for SQLSTAT and SQLSTATE, respectively.)
- No declaration for either SQLSTATE or SQLCODE: DB2 assigns the value in SQLCADE to SQLCODE.

If the precompiler or coprocessor encounters an INCLUDE SQLCA statement, it ignores the statement and issues a warning message. The precompiler or coprocessor also does not recognize hand-coded definitions, and a hand-coded definition creates a compile-time conflict with the generated definition. A similar conflict arises if definitions of SQLCADE or SQLSTAT, other than the ones generated by the DB2 precompiler or coprocessor, appear in the program.

Positioned updates of columns

Certain SQL processing options affect the use of the FOR UPDATE clause to achieve positioned column updates.

The NOFOR SQL processing option affects the use of the FOR UPDATE clause. The NOFOR option is in effect when either of the following are true:

- The NOFOR option is specified.
- The STDSQL(YES) option is in effect.

Otherwise, the NOFOR option is not in effect. The following table summarizes the differences when the option is in effect and when the option is not in effect:

Table 57. The NOFOR SQL processing option

When NOFOR is in effect	When NOFOR is not in effect
The use of the FOR UPDATE clause in the SELECT statement of the DECLARE CURSOR statement is optional. This clause restricts updates to the specified columns and causes the acquisition of update locks when the cursor is used to fetch a row. If no columns are specified, positioned updates can be made to any updatable columns in the table or view that is identified in the first FROM clause in the SELECT statement. If the FOR UPDATE clause is not specified, positioned updates can be made to any columns that the program has DB2 authority to update.	The FOR UPDATE clause must be specified.

Table 57. The NOFOR SQL processing option (continued)

When NOFOR is in effect	When NOFOR is not in effect
DBRMs must be built entirely in virtual storage, which might possibly increase the virtual storage requirements of the DB2 precompiler or coprocessor. However, creating DBRMs entirely in virtual storage might cause concurrency problems with DBRM libraries.	DBRMs can be built incrementally using the DB2 precompiler or coprocessor.

SQL processing options do not affect ODBC behavior.

Mappings from SQL to XML

DB2 maps SQL to XML data according to industry standards and performs several different mappings.

To construct XML data from SQL data, the following mappings are performed:
- SQL character sets to XML character sets
- SQL identifiers to XML names
- SQL data values to XML data values

DB2 maps SQL to XML data according to industry standards. For complete information, see *Information technology - Database languages - SQL- Part 14: XML-Related Specifications (SQL/XML) ISO/IEC 9075-14:2003*.

Mapping SQL character sets to XML character sets

The character set used for XML data is Unicode UTF-8. SQL character data is converted into Unicode when it is used in XML built-in functions.

Mapping SQL identifiers to XML names

Many SQL identifiers that contain certain characters must be escaped when the SQL identifier is converted into an XML name.

Strings that start with 'XML', in any case combination, are reserved for standardization, and characters such as '#', '{', and '}' are not allowed in XML names. Many SQL identifiers containing these characters have to be escaped when converting into XML names.

Full escaping is applied to SQL identifiers that are column names to derive an XML name. The mapping converts a colon (:) to _x003A_, _x to _X005F_x, and other restricted characters to a string of the form *_xUUUU_* where *xUUUU_* is the Unicode value for the character. An identifier with an initial 'xml' (in any case combination) is escaped by mapping the initial 'x' or 'X' to _x0058_ or _0078_, respectively, while the partially escaped variant does not.

Mapping SQL data values to XML data values

SQL data values are mapped to XML values based on SQL data types.

The following data types are not supported and cannot be used as arguments to XML value constructors:
- ROWID
- Character strings that are defined with the FOR BIT DATA attribute

- Binary strings
- A string or a binary string distinct type that is based on a ROWID, FOR BIT DATA character string, or BLOB

For supported data types, the encoding scheme for XML values is Unicode.

Chapter 3. Functions

A *function* is an operation denoted by a function name followed by zero or more input values that are enclosed in parentheses. It represents a relationship between a set of input values and a set of result values. The input values to a function are called *arguments*.

The types of functions are aggregate, scalar, and table. A built-in function is classified as a aggregate function or a scalar function. A user-defined function can be a column, scalar, or table function.

If a column mask is used to mask the column values in the final result table and a column mask is applied to a column that is an argument for a function, the result of the function might be different because the column mask is applied to the column before the function operation can take place. For example, applying a column mask to column SSN can change the result of the aggregate function, COUNT(DISTINCT SSN). The DISTINCT operation is performed on the masked column values.

OLAP specification and functions
The RANK, DENSE_RANK, and ROW_NUMBER specifications are sometimes referred to as built-in 'functions'. Refer to "OLAP specification" on page 282 for more information on these specifications.

DB2 MQSeries functions
DB2 MQSeries functions integrate MQSeries messaging operations within SQL statements. The functions help you integrate MQSeries messaging with database applications. You can use the functions to access MQSeries messaging from within SQL statements and to combine MQSeries messaging with DB2 database access.

The functions can be scalar or table functions. For more information on using MQSeries functions, see the information on enabling MQSeries functions in *DB2 Installation Guide* and on programming techniques in *DB2 Application Programming and SQL Guide*.

Administrative task scheduler functions
The administrative task scheduler table functions provide information and status about the tasks that are scheduled to run using the administrative task scheduler. The administrative task scheduler provides the ability to run stored procedures, JCL jobs, and other administrative tasks according to a time or an event-based schedule. Refer to *DB2 Administration Guide* for additional information about the administrative task scheduler.

The following table lists the functions that DB2 supports.

Table 58. Supported functions

Function name	Description
ABS	Returns the absolute value of its argument
ACOS	Returns the arc cosine of an argument as an angle, expressed in radians
ADD_MONTHS	Returns a date that represents the date argument plus the number of months argument

Table 58. Supported functions (continued)

Function name	Description
ADMIN_TASK_LIST	Returns a table with one row for each of the tasks that are defined in the administrative task scheduler task list
ADMIN_TASK_STATUS	Returns a table with one row for each task in the administrative task scheduler task list that contains the status for the last time the task was run
ASCII	Returns the ASCII code value of the leftmost character of the argument as an integer
ASCII_CHR	Returns the character that corresponds to the ASCII code value that is specified by the argument
ASCII_STR	Returns an ASCII version of the character or graphic string argument.
ASIN	Returns the arc sine of an argument as an angle, expressed in radians
ATAN	Returns the arc tangent of an argument as an angle, expressed in radians
ATANH	Returns the hyperbolic arc tangent of an argument as an angle, expressed in radians
ATAN2	Returns the arc tangent of *x* and *y* coordinates as an angle, expressed in radians
AVG	Returns the average of a set of numbers
BLOB	Returns a BLOB representation of its argument
BIGINT	Returns a big integer representation of its argument
BITAND, BITANDNOT, BITOR, BITXOR, and BITNOT	Return a corresponding base 10 integer value in a data type that is based on the data type of the input arguments.
BINARY	Returns a fixed-length binary string representation of its argument
CCSID_ENCODING	Returns the encoding scheme of a CCSID with a value of ASCII, EBCDIC, UNICODE, or UNKNOWN
CEILING	Returns the smallest integer greater than or equal to the argument
CHAR	Returns a fixed-length character string representation of its argument
CHARACTER_LENGTH	Returns the length of its argument in the number of string units that are specified
CLOB	Returns a CLOB representation of its argument
COALESCE	Returns the first argument in a set of arguments that is not null
COLLATION_KEY	Returns a string that represents the collation key of the argument in the specified collation
COMPARE_DECFLOAT	Returns a SMALLINT value that indicates whether two arguments are equal, or unordered, or whether one argument is greater than the other.
CONCAT	Returns the concatenation of two strings
CONTAINS	Returns a result about whether or not a match was found during a search of a text search index
CORRELATION	Returns the coefficient of the correlation of a set of number pairs
COS	Returns the cosine of an argument that is expressed as an angle in radians
COSH	Returns the hyperbolic cosine of an argument that is expressed as an angle in radians
COUNT	Returns the number of rows or values in a set of rows or values

Table 58. Supported functions (continued)

Function name	Description
COUNT_BIG	Same as COUNT, except the result can be greater than the maximum value of an integer
COVARIANCE or COVARIANCE_SAMP	Returns the (population) covariance of a set of number pairs
DATE	Returns a date derived from its argument
DAY	Returns the day part of its argument
DAYOFMONTH	Similar to DAY
DAYOFWEEK	Returns an integer in the range of 1 to 7, where 1 represents Sunday
DAYOFWEEK_ISO	Returns an integer in the range of 1 to 7, where 1 represents Monday
DAYOFYEAR	Returns an integer in the range of 1 to 366, where 1 represents January 1
DAYS	Returns an integer representation of a date
DBCLOB	Returns a DBCLOB representation of its argument
DECIMAL or DEC	Returns a decimal representation of its argument
DECFLOAT	Returns a DECFLOAT representation of its argument
DECFLOAT_FORMAT	Returns a DECFLOAT(34) value that is based on the interpretation of the input string using the specified format.
DECFLOAT_SORTKEY	Returns a binary value that can be used when sorting DECFLOAT values
DECODE	Returns a specified *result-expression* based on a comparison of input expressions (similar to the CASE expression).
DECRYPT_BINARY, DECRYPT_BIT, DECRYPT_CHAR, or DECRYPT_DB	Returns the decrypted value of an encrypted argument
DEGREES	Returns the number of degrees for an argument that is expressed in radians
DIFFERENCE	Returns a value that represents the difference between the sounds of two strings based on applying the SOUNDEX function to the strings.
DIGITS	Returns a character string representation of a number
DOUBLE or DOUBLE_PRECISION	Returns a double precision floating-point representation of its argument
DSN_XMLVALIDATE	Returns an XML value that is the result of applying XML schema validation to the first argument.
EBCDIC_CHR	Returns the character that corresponds to the EBCDIC code value that is specified by the argument
EBCDIC_STR	Returns an EBCDIC version of the string argument
ENCRYPT_TDES	Returns the argument as an encrypted value
EXP	Returns the exponential function of an argument
EXTRACT	Returns a portion of a date or timestamp based on its arguments
FLOAT	Same as DOUBLE
FLOOR	Returns the largest integer that is less than or equal to the argument
GENERATE_UNIQUE	Returns a character string of bit data that is unique compared to any other execution of the function
GETHINT	Returns the embedded password hint from encrypted data, if one exists

Table 58. Supported functions (continued)

Function name	Description
GETVARIABLE	Returns a varying-length character string representation of the value of a session variable
GRAPHIC	Returns a fixed-length graphic string representation of its argument
HEX	Returns a hexadecimal representation of its argument
HOUR	Returns the hour part of its argument
IDENTITY_VAL_LOCAL	Returns the most recently assigned value for an identity column
IFNULL	Returns the first argument in a set of two arguments that is not null
INSERT	Returns a string that is composed of an argument inserted into another argument at the same position where some number of bytes have been deleted
INTEGER or INT	Returns an integer representation of its argument
JULIAN_DAY	Returns an integer that represents the number of days from January 1, 4712 B.C.
LAST_DAY	Returns a date that represents the last day of the month of the date argument
LCASE	Returns a string with the characters converted to lowercase
LEFT	Returns a string that consists of the specified number of leftmost bytes or the specified string units
LENGTH	Returns the length of its argument
LN	Returns the natural logarithm of an argument
LOCATE	Returns the starting position of one string within another string
LOCATE_IN_STRING	Returns the starting position of the first occurrence of one string within another string
LOG10	Returns the base 10 logarithm of an argument
LOWER	Returns a string with the characters converted to lowercase
LPAD	Returns a string that is padded on the left with blanks or a specified string
LTRIM	Returns the characters of a string with the leading blanks or hexadecimal zeros removed
MAX (aggregate)	Returns the maximum value in a set of column values
MAX (scalar)	Returns the maximum value in a set of values
MICROSECOND	Returns the microsecond part of its argument
MIDNIGHT_SECONDS	Returns an integer in the range of 0 to 86400 that represents the number of seconds between midnight and the argument
MIN (aggregate)	Returns the minimum value in a set of column values
MIN (scalar)	Returns the minimum value in a set of values
MINUTE	Returns the minute part of its argument
MOD	Returns the remainder of one argument divided by a second argument
MONTH	Returns the month part of its argument
MONTHS_BETWEEN	Returns an estimate of the number of months between two arguments
MQREAD	Returns a message from a specified MQSeries location (return value of VARCHAR) without removing the message from the queue

Table 58. Supported functions (continued)

Function name	Description
MQREADALL	Returns a table containing the messages and message metadata from a specified MQSeries location with a VARCHAR column and without removing the messages from the queue
MQREADALLCLOB	Returns a table containing the messages and message metadata from a specified MQSeries location with a CLOB column and without removing the messages from the queue
MQREADCLOB	Returns a message from a specified MQSeries location (return value of CLOB) without removing the message from the queue
MQRECEIVE	Returns a message from a specified MQSeries location (return value of VARCHAR) with removal of message from the queue
MQRECEIVEALL	Returns a table containing the messages and message metadata from a specified MQSeries location with a VARCHAR column and with removal of messages from the queue
MQRECEIVEALLCLOB	Returns a table containing the messages and message metadata from a specified MQSeries location with a CLOB column and with removal of messages from the queue
MQRECEIVECLOB	Returns a message from a specified MQSeries location (return value of CLOB) with removal of message from the queue
MQSEND	Sends data to a specified MQSeries location, and returns a varying-length character string that indicates whether the function was successful or unsuccessful
MULTIPLY_ALT	Returns the product of the two arguments as a decimal value, used when the sum of the argument precisions exceeds 31
NEXT_DAY	Returns a timestamp that represents the first weekday, specified by the second argument, after the date argument
NORMALIZE_DECFLOAT	Returns a DECFLOAT value that is the result of normalizing the input argument
NORMALIZE_STRING	Returns a string value that is the result of normalizing the input Unicode value
NULLIF	Returns NULL if the arguments are equal; else the first argument
NVL	Returns the first argument that is not null.
OVERLAY	Returns a string that is composed of an argument inserted into another argument at the same position where some number of bytes have been deleted
POSITION	Returns the position of the first occurrence of an argument within another argument where the position is expressed in terms of the string units that are specified
POSSTR	Returns the position of the first occurrence of an argument within another argument
POWER	Returns the value of one argument raised to the power of a second argument
QUANTIZE	Returns a DECFLOAT value that is equal in value (except for any rounding) and sign to the first argument and which has an exponent set to be equal to the exponent of the second argument
QUARTER	Returns an integer in the range of 1 to 4 that represents the quarter of the year for the date specified in the argument
RADIANS	Returns the number of radians for an argument that is expressed in degrees

Table 58. Supported functions (continued)

Function name	Description
RAISE_ERROR	Raises an error in the SQLCA with the specified SQLSTATE and error description
RAND	Returns a double precision floating-point random number
REAL	Returns a single precision floating-point representation of its argument
REPEAT	Returns a character string composed of an argument repeated a specified number of times
REPLACE	Returns a string in which all occurrences of an argument within a second argument are replaced with a third argument
RID	Returns the RID of a row
RIGHT	Returns a string that consists of the specified number of rightmost bytes or specified string unit
ROUND	Returns a number rounded to the specified number of places to the right or left of the decimal place
ROUND_TIMESTAMP	Returns a timestamp rounded to the unit specified by the timestamp format string
ROWID	Returns a row ID representation of its argument
RPAD	Returns a string that is padded on the right with blanks or a specified string
RTRIM	Returns the characters of an argument with the trailing blanks or hexadecimal zeros removed
SCORE	Returns a relevance score that measures how well a document matches the query used to search a text search index
SECOND	Returns the second part of its argument
SIGN	Returns the sign of an argument
SIN	Returns the sine of an argument that is expressed as an angle in radians
SINH	Returns the hyperbolic sine of an argument that is expressed as an angle in radians
SMALLINT	Returns a small integer representation of its argument
SOAPHTTPC or SOAPHTTPV	Returns a CLOB or VARCHAR representation of XML data from a request to a web service
SOAPHTTPNC or SOAPHTTPNV	Returns a complete CLOB or VARCHAR representation of XML data from a complete request to a web service
SOUNDEX	Returns a value that represents the sound of the words in the argument.
SPACE	Returns a string that consists of the number of blanks the argument specifies
SQRT	Returns the square root of its argument
STDDEV or STDDEV_SAMP	Returns the standard deviation (/n), or the sample standard deviation (/n-1), of a set of numbers
STRIP	Returns the characters of a string with the blanks (or specified character) at the beginning, end, or both beginning and end of the string removed
SUBSTR	Returns a substring of a string
SUBSTRING	Returns a substring of a string using the specified string units

Table 58. Supported functions (continued)

Function name	Description
SUM	Returns the sum of a set of numbers
TAN	Returns the tangent of an argument that is expressed as an angle in radians
TANH	Returns the hyperbolic tangent of an argument that is expressed as an angle in radians
TIME	Returns a time derived from its argument
TIMESTAMP	Returns a timestamp derived from its arguments
TIMESTAMPADD	Returns a timestamp derived from adding the specified interval to a timestamp
TIMESTAMP_FORMAT	Returns a timestamp for a character string expression, using a specified format to interpret the string
TIMESTAMP_ISO	Returns a timestamp derived from its arguments
TIMESTAMPDIFF	Returns an estimated number of the specified intervals based on the difference between two timestamps
TIMESTAMP_TZ	Returns a timestamp with a time zone derived from its arguments
TO_CHAR	Returns a character string representation of a timestamp value that has been formatted using a specified character template.
TO_DATE	Returns a timestamp value that is based on the interpretation of the input string using the specified format.
TO_NUMBER	Returns a DECFLOAT(34) value that is based on the interpretation of the input string using the specified format.
TOTALORDER	Returns a SMALLINT value that indicates the comparison order of two arguments
TRANSLATE	Returns a string with one or more characters translated
TRUNCATE	Returns a number truncated to the specified number of places to the right or left of the decimal point
TRUNC_TIMESTAMP	Returns a timestamp truncated to the unit specified by the timestamp format string
UCASE	Returns a string with the characters converted to uppercase
UNICODE	Returns the Unicode (UTF-16) code value of the leftmost character of the argument as an integer
UNICODE_STR	Returns a string in Unicode (UTF-8 or UTF-16) that represents a Unicode encoding of the argument
UPPER	Returns a string with the characters converted to uppercase
VALUE	Same as COALESCE
VARBINARY	Returns a varying-length binary string representation of its argument
VARCHAR	Returns the varying-length character string representation of its argument
VARCHAR_FORMAT	Returns a varying-length character string representation of a timestamp, with the string in a specified format
VARGRAPHIC	Returns a varying-length graphic string representation of its argument
VARIANCE or VARIANCE_SAMP	Returns the variance, or sample variance, of a set of numbers

Table 58. Supported functions (continued)

Function name	Description
VERIFY_GROUP_FOR_USER	Returns a value that indicates whether the primary authorization ID and the group authorization IDs that are associated with the first argument are included in the authorization names that are specified in the list of the second argument.
VERIFY_ROLE_FOR_USER	Returns a value that indicates whether the roles that are associated with the first argument are included in the role names that are specified in the list of the second argument.
VERIFY_TRUSTED_CONTEXT_FOR_USER	Returns a value that indicates whether the authorization ID that is associated with first argument has acquired a role in a trusted connection and whether that acquired role is included in the role names that are specified in the list of the second argument.
WEEK	Returns an integer that represents the week of the year with Sunday as the first day of the week
WEEK_ISO	Returns an integer that represents the week of the year with Monday as first day of a week
XMLAGG	Returns an XML type that represents a concatenation of XML elements from a collection of XML elements
XMLATTRIBUTES	Returns an XML sequence that contains an XQuery attribute node for each non-null argument
XMLCOMMENT	Returns an XML value with a single comment node from a string expression
XMLCONCAT	Returns an XML value that represents a forest of XML elements generated by concatenating a variable number of arguments
XMLDOCUMENT	Returns an XML value with a single document node and zero or more nodes as its children
XMLELEMENT	Returns an XML value that represents an XML element
XMLFOREST	Returns an XML value that represents a forest of XML elements that all share a specific pattern
XMLMODIFY	Returns an XML value that might have been modified by the evaluation of an XQuery updating expression and XQuery variables that are specified as input arguments.
XMLNAMESPACES	Returns the declaration of one or more XML namespaces
XMLPARSE	Returns an XML value from parsing the argument as an XML document
XMLPI	Returns an XML value with a single processing instruction node
XMLQUERY	Returns an XML value from the evaluation of an XPath expression against a set of arguments
XMLSERIALIZE	Returns an SQL character string or a BLOB value from an XML value
XMLTABLE	Returns a result table from the evaluation of XQuery expressions, possibly using specified input arguments as XQuery variables
XMLTEXT	Returns an XML value with a single text node that contains the value of the argument
YEAR	Returns the year part of its argument

Aggregate functions

An aggregate function receives a set of values for each argument (such as the values of a column) and returns a single-value result for the set of input values. Certain rules apply to all aggregate functions.

The following information applies to all aggregate functions, except for the COUNT(*) and COUNT_BIG(*), variations of the COUNT and COUNT_BIG functions, and the XMLAGG function.

The argument of an aggregate function is a set of values derived from an expression. The expression must not include another aggregate function or a scalar fullselect. The scope of the set is a group or an intermediate result table, as explained in the information on the GROUP BY clause.

If a GROUP BY clause is specified in a query and the intermediate result from the FROM, WHERE, GROUP BY, and HAVING clauses is the empty set, then the aggregate functions are not applied and the result of the query is the empty set.

If the GROUP BY clause is not specified in a query and the intermediate result table of the FROM, WHERE, and HAVING clauses is the empty set, then the aggregate functions are applied to the empty set.

For example, the result of the following SELECT statement is the number of distinct values of JOB for employees in department D11:

```
SELECT COUNT(DISTINCT JOB)
  FROM DSN8B10.EMP
  WHERE WORKDEPT = 'D11';
```

The keyword DISTINCT is not an argument of the function but rather a specification of an operation that is performed before the function is applied. If DISTINCT is specified, redundant duplicate values are eliminated. If ALL is implicitly or explicitly specified, redundant duplicate values are not eliminated. DISTINCT must not be specified preceding an XML value.

For compatibility with other SQL implementations, UNIQUE can be specified as a synonym for DISTINCT in aggregate functions.

When interpreting the DISTINCT clause for decimal floating-point values that are numerically equal, the number of significant digits in the value is not considered. For example, the decimal floating-point number 123.00 is not distinct from the decimal floating-point number 123. The representation of the number returned from the query will be any one of the representations encountered (for example, either 123.00 or 123).

An aggregate function can be used in a WHERE clause only if that clause is part of a subquery of a HAVING clause and the column name specified in the expression is a correlated reference to a group. If the expression includes more than one column name, each column name must be a correlated reference to the same group.

The result of the COUNT and COUNT_BIG functions cannot be the null value. As specified in the description of AVG, MAX, MIN, STDDEV, SUM, and VARIANCE, the result is the null value when the function is applied to an empty set. However, the result is also the null value when the function is specified in an outer select

list, the argument is given by an arithmetic expression, and any evaluation of the expression causes an arithmetic exception (such as division by zero).

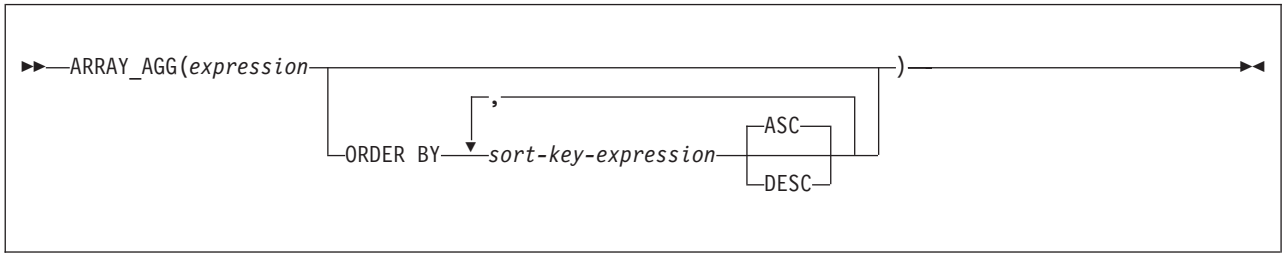
If the argument values of an aggregate function are strings from a column with a field procedure, the function is applied to the encoded form of the values and the result of the function inherits the field procedure.

Related reference:

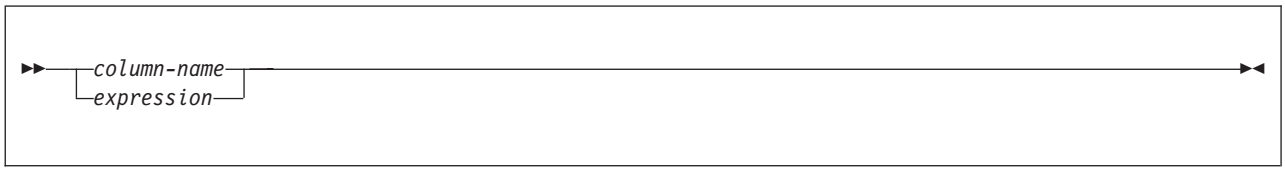
"group-by-clause" on page 797

ARRAY_AGG

The ARRAY_AGG function returns an array in which each value of the input set is assigned to an element of the array.



sort-key-expression



The schema is SYSIBM.

expression
Specifies an expression that returns a value with a data type that is valid for an array element. The data type of the expression must be a data type that can be specified in a CREATE TYPE (array) statement.

ORDER BY
Specifies the order of the rows from the same grouping set that are processed in the aggregation. If the ORDER BY clause is not specified, or if the ORDER BY clause cannot differentiate the order of the sort key value, the rows in the same grouping set are arbitrarily ordered.

sort-key-expression
Specifies a sort key value that is either a column name or an expression.

If the sort key value is a constant, the constant does not refer to the position of the output column, but is simply a constant, which implies that there is no sort key.

ASC Processes the sort key in ascending order. This is the default option.

DESC Processes the sort key in descending order.

The result data type of ARRAY_AGG is an array. The result is an ordinary array. The data type of an array element of the result array is the same as the type of *expression*.

If the result of an ARRAY_AGG function is assigned to a target variable through a SELECT INTO statement, the result of the ARRAY_AGG function is converted to the data type of the target variable.

If a SELECT clause includes multiple invocations of the ARRAY_AGG function, all invocations of ARRAY_AGG in the SELECT clause that explicitly specify an ORDER BY clause must specify the same order.

The ARRAY_AGG function can be invoked only in SQL PL, in the following contexts:

- The SELECT list of a SELECT INTO statement
- The SELECT list of the outermost fullselect in the definition of a cursor that is not scrollable
- The SELECT list of a scalar subquery that provides a source value for a SET *assignment-statement* or SQL PL *assignment-statement*
- A RETURN statement in an SQL scalar function

The following restrictions apply to ARRAY_AGG:

- ARRAY_AGG cannot be used as part of an OLAP specification.
- A fullselect that contains an invocation of ARRAY_AGG cannot contain an ORDER BY clause.
- A fullselect that contains an invocation of ARRAY_AGG cannot contain a DISTINCT keyword in its SELECT list.
- The SELECT clause or HAVING clause of the fullselect that contains an invocation of ARRAY_AGG cannot contain a subquery.
- A SELECT clause that includes an invocation of the ARRAY_AGG function that returns an array of LOBs must not also include a GROUP BY clause.
- A SELECT clause that includes an invocation of the ARRAY_AGG function must not also include an invocation of the XMLAGG function.

Example 1: Use ARRAY_AGG in an assignment statement to assign the values of the DECIMALARRAY array to the array INTARRAY.

```
SET INTARRAY = (SELECT ARRAY_AGG(VAL) FROM UNNEST(DECIMALARRAY) AS T(VAL));
```

Example 2: Use ARRAY_AGG in a SELECT INTO statement to assign the values of the ESALARIES array to the array ARRAY2.

```
SELECT ARRAY_AGG(T.VAL) INTO ARRAY2 FROM UNNEST(ESALARIES) AS T(VAL);
```

Example 3: Suppose that user-defined type PHONELIST and table EMPLOYEE have the following definitions:

```
CREATE TYPE PHONELIST AS DECIMAL(10,0) ARRAY[10];

CREATE TABLE EMPLOYEE ( ID INTEGER NOT NULL,
 PRIORITY INTEGER NOT NULL,
 PHONENUMBER DECIMAL(10,0),
 PRIMARY KEY(ID, PRIORITY)) ;
```

The following SQL PL procedure uses a SELECT INTO statement that returns a list of contact numbers under which an employee can be reached, ordered by priority.

```
CREATE PROCEDURE GETPHONENUMBERS
 (IN EMPID INTEGER,
 OUT NUMBERS PHONELIST)
 BEGIN
  SELECT ARRAY_AGG(PHONENUMBER ORDER BY PRIORITY)
   INTO NUMBERS
   FROM EMPLOYEE
  WHERE ID = EMPID;
 END
```

The following SQL PL procedure uses SET *assignment-statement* to return the list of contact numbers in an arbitrary order.

```
CREATE PROCEDURE GETPHONENUMBERS
 (IN EMPID INTEGER,
 OUT NUMBERS PHONELIST)
 BEGIN
  SET NUMBERS =
   (SELECT ARRAY_AGG(PHONENUMBER)
     FROM EMPLOYEE
     WHERE ID = EMPID);
 END
```

AVG

The AVG function returns the average of a set of numbers.

The schema is SYSIBM.

The argument values can be of any built-in numeric data type, and their sum must be within the range of the data type of the result.

The arguments can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34).

The data type of the result is determined as follows:

- DECFLOAT(34) if the argument is DECFLOAT(*n*).
- Large integer is the argument is small integer.
- Double precision floating-point is the argument is single precision floating-point.
- Otherwise, the result is the same as the data type of the argument.

The result can be null.

If the data type of the argument values is decimal with precision *p* and scale *s*, the precision (P) and scale (S) of the result depend on *p* and the decimal precision option:

- If *p* is greater than 15 or the DEC31 option is in effect, P is 31 and S is `max(0,28-p+s)`.
- Otherwise, P is 15 and S is 15-*p*+*s*.

The function is applied to the set of values derived from the argument values by the elimination of null values. If DISTINCT is specified, redundant duplicate values are also eliminated.

If the function is applied to an empty set, the result is the null value. Otherwise, the result is the average value of the set. The order in which the summation part of the operation is performed is undefined but every intermediate result must be within the range of the result data type.

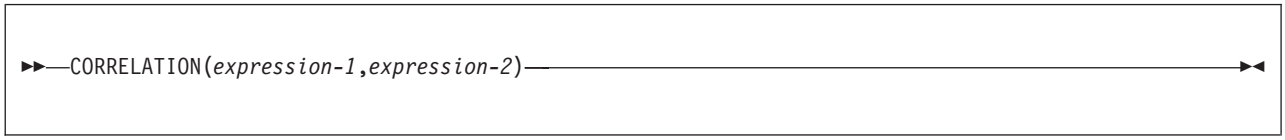
If the type of the result is integer, the fractional part of the average is lost.

Example: Assuming DEC15, set the DECIMAL(15,2) variable AVERAGE to the average salary in department D11 of the employees in the sample table DSN8B10.EMP.

```
EXEC SQL SELECT AVG(SALARY)
  INTO :AVERAGE
  FROM DSN8B10.EMP
  WHERE WORKDEPT = 'D11';
```

CORRELATION

The CORRELATION function returns the coefficient of the correlation of a set of number pairs.



The schema is SYSIBM.

The argument values must each be the value of any built-in numeric data type.

If an argument is DECFLOAT(*n*), the result of the function is DECFLOAT(34). Otherwise, the result of the function is double precision floating-point. The result is between -1 and 1. The result can be null.

The function is applied to the set of (*expression-1*, *expression-2*) pairs derived from the argument values by the elimination of all pairs for which either *expression-1* or *expression-2* is null.

If the function is applied to an empty set, or if either STDDEV(*expression-1*) or STDDEV(*expression-2*) is equal to zero, the result is a null value. Otherwise, the result is the correlation coefficient for the value pairs in the set. The result is equivalent to the following expression:

```
COVARIANCE(expression-1,expression-2)/
  (STDDEV(expression-1)* STDDEV(expression-2))
```

The order in which the values are aggregated is undefined, but every intermediate result must be within the range of the result data type.

CORR can be specified as a synonym for CORRELATION.

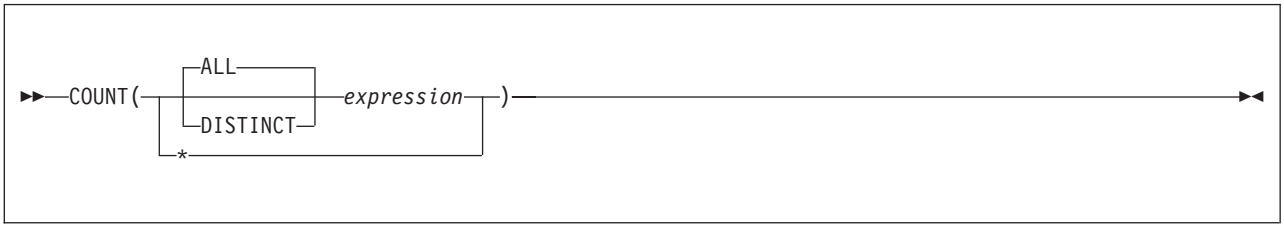
Example: Using sample table DSN8B10.EMP, set the host variable :corrln (double-precision floating point) to the correlation between the salary and the bonus for those employees in department (WORKDEPT) 'A00'.

```
SELECT CORRELATION(SALARY, BONUS) INTO :corrln
    FROM DSN8B10.EMP WHERE WORKDEPT = 'A00';
```

:corrln is set to approximately 9.99853953399538E-001.

COUNT

The COUNT function returns the number of rows or values in a set of rows or values.



The schema is SYSIBM.

The argument values can be of any built-in data type other than a BLOB, CLOB, DBCLOB, or XML.

The result is a large integer. The result cannot be null.

The argument of `COUNT(*)` is a set of rows. The result is the number of rows in the set. Any row that includes only null values is included in the count.

The argument of `COUNT(expression)` or `COUNT(ALL expression)` is a set of values. The function is applied to the set of values derived from the argument values by the elimination of null values. The result is the number of nonnull values in the set, including duplicates.

The argument of `COUNT(DISTINCT expression)` is a set of values. The function is applied to the set of values derived from the argument values by the elimination of null values and redundant duplicate values. The result is the number of different nonnull values in the set.

Example 1: Set the integer host variable FEMALE to the number of females represented in the sample table DSN8B10.EMP.

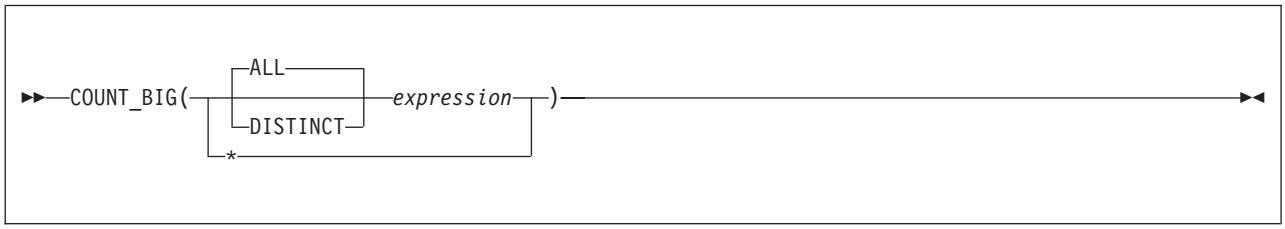
```
EXEC SQL SELECT COUNT(*)
  INTO :FEMALE
  FROM DSN8B10.EMP
  WHERE SEX = 'F';
```

Example 2: Set the integer host variable FEMALE_IN_DEPT to the number of departments that have at least one female as a member.

```
EXEC SQL SELECT COUNT(DISTINCT WORKDEPT)
  INTO :FEMALE_IN_DEPT
  FROM DSN8B10.EMP
  WHERE SEX = 'F';
```

COUNT_BIG

The COUNT_BIG function returns the number of rows or values in a set of rows or values. It is similar to COUNT except that the result can be greater than the maximum value of an integer.



The schema is SYSIBM.

The argument values can be of any built-in data type other than a BLOB, CLOB, DBCLOB, or XML.

The result of the function is a decimal number with precision 31 and scale 0. The result cannot be null.

The argument of COUNT_BIG(*) is a set of rows. The result is the number of rows in the set. A row that includes only null values is included in the count.

The argument of COUNT_BIG(*expression*) or COUNT_BIG(ALL *expression* is a set of values. The function is applied to the set of values derived from the argument values by the elimination of null values. The result is the number of nonnull values in the set, including duplicates.

The argument of COUNT_BIG(DISTINCT *expression*) is a set of values. The function is applied to the set of values derived from the argument values by the elimination of null and redundant duplicate values. The result is the number of different nonnull values in the set.

Example 1: Set the integer host variable FEMALE to the number of females represented in the sample table DSN8B10.EMP.

```
EXEC SQL SELECT COUNT_BIG(*)
  INTO :FEMALE
  FROM DSN8B10.EMP
  WHERE SEX = 'F';
```

Example 2: Set the integer host variable FEMALE_IN_DEPT to the number of departments that have at least one female as a member.

```
EXEC SQL SELECT COUNT_BIG(DISTINCT WORKDEPT)
  INTO :FEMALE_IN_DEPT
  FROM DSN8B10.EMP
  WHERE SEX = 'F';
```

Example 3: To create a sourced function that is similar to the built-in COUNT_BIG function, the definition of the sourced function must include the type of the column that can be specified when the new function is invoked. In this example, the CREATE FUNCTION statement creates a sourced function that takes a CHAR column as input and uses COUNT_BIG to perform the counting. The result is

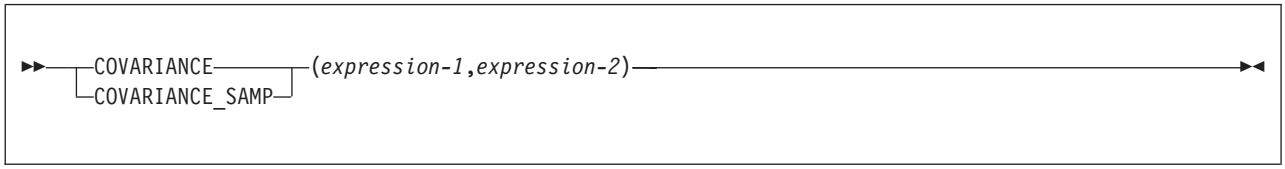
returned as a double precision floating-point number. The query shown counts the number of unique departments in the sample employee table.

```
CREATE FUNCTION RICK.COUNT(CHAR()) RETURNS DOUBLE
       SOURCE SYSIBM.COUNT_BIG(CHAR());
SET CURRENT PATH RICK, SYSTEM PATH;
SELECT COUNT(DISTINCT WORKDEPT) FROM DSN8B10.EMP;
```

The empty parenthesis in the parameter list for the new function (RICK.COUNT) means that the input parameter for the new function is the same type as the input parameter for the function named in the SOURCE clause. The empty parenthesis in the parameter list in the SOURCE clause (SYSIBM.COUNT_BIG) means that the length attribute of the CHAR parameter of the COUNT_BIG function is ignored when DB2 locates the COUNT_BIG function.

COVARIANCE or COVARIANCE_SAMP

The COVARIANCE and COVARIANCE_SAMP functions return the covariance (population) of a set of number pairs.



The schema is SYSIBM.

The argument values must each be the value of any built-in numeric data type.

If an argument is DECFLOAT(*n*), the result of the function is DECFLOAT(34). Otherwise, the result of the function is double precision floating-point. The result can be null.

The function is applied to the set of (*expression-1*, *expression-2*) pairs that are derived from the argument values by the elimination of all pairs for which either *expression-1* or *expression-2* is null.

If the function is applied to an empty set, the result is a null value. Otherwise, the result is the covariance of the value pairs in the set. The result is equivalent to the following outputs:

For COVARIANCE:

1. Let *avgexp1* be the result of `AVG(expression-1)` and let *avgexp2* be the result of `AVG(expression-2)`.
2. The result of `COVARIANCE(expression-1,expression-2)` is `AVG( (expression-1 - avgexp1) * (expression-2 - avgexp2))`

For COVARIANCE_SAMP:

1. Let *samp_avgexp1* be the result of `SUM(expression-1)/n-1` and let *samp_avgexp2* be the result of `SUM(expression-2)/n-1`.
2. The result of `COVARIANCE_SAMP(expression-1, expression-2)` is `AVG((expression-1 - samp_avgexp1) * (expression-2 - samp_avgexp2))`

The order in which the values are aggregated is undefined, but every intermediate result must be within the range of the result data type.

COVAR can be specified as a synonym for COVARIANCE.

COVAR_SAMP can be specified as a synonym for COVARIANCE_SAMP.

Example: Using sample table DSN8B10.EMP, set the host variable *covarnce* (double-precision floating point) to the covariance between the salary and the bonus for those employees in department (WORKDEPT) 'A00'.

```
SELECT COVARIANCE(SALARY, BONUS) INTO :covarnce
  FROM EMPLOYEE WHERE WORKDEPT = 'A00';
```

covarnce is set to approximately 1.68888888888889E+006.

MAX

The MAX function returns the maximum value in a set of values.

The schema is SYSIBM.

The arguments must be compatible. For more information on compatibility, refer to the compatibility matrix in Table 23 on page 121. All arguments except the first argument can be parameter markers.

This function cannot be used as a source function when creating a user-defined function.

expression
An expression that returns the value of a built-in data type. Each expression must return a value that is not a CLOB, DBCLOB, BLOB, ROWID, or XML. Character string arguments and binary string arguments cannot have a length attribute greater than 32704, and graphic string arguments cannot have a length attribute greater than 16352.

If there are any mixed character string or graphic string and numeric arguments, the string value is implicitly cast to a DECFLOAT(34) value.

The result of the function is the largest argument value. The data type of the result and its other attributes (for example, the length and CCSID of a string or a datetime value) are the same as the data type and attributes of the argument values. The result can be null.

The function is applied to the set of values derived from the argument values by the elimination of null values.

If the function is applied to an empty set, the result is the null value. Otherwise, the result is the maximum value in the set.

The specification of DISTINCT has no effect on the result and is not advised.

Example 1: Set the DECIMAL(8,2) variable MAX_SALARY to the maximum monthly salary of the employees represented in the sample table DSN8B10.EMP.

```
EXEC SQL SELECT MAX(SALARY) / 12
  INTO :MAX_SALARY
  FROM DSN8B10.EMP;
```

Example 2: Find the surname that comes last in the collating sequence for the employees represented in the sample table DSN8B10.EMP. Set the VARCHAR(15) variable *LAST_NAME* to that surname.

```
EXEC SQL SELECT MAX(LASTNAME)
  INTO :LAST_NAME
  FROM DSN8B10.EMP;
```

MIN

The MIN function returns the minimum value in a set of values.

The schema is SYSIBM.

The arguments must be compatible. For more information on compatibility, refer to the compatibility matrix in Table 23 on page 121. All arguments except the first argument can be parameter markers.

This function cannot be used as a source function when creating a user-defined function.

expression
An expression that returns the value of a built-in data type. Each expression must return a value that is not a CLOB, DBCLOB, BLOB, ROWID, or XML. Character string arguments and binary string arguments cannot have a length attribute greater than 32704, and graphic string arguments cannot have a length attribute greater than 16352.

If there are any mixed character string or graphic string and numeric arguments, the string value is implicitly cast to a DECFLOAT(34) value.

The result of the function is the smallest argument value. The data type of the result and its other attributes (for example, the length and CCSID of a string or a datetime value) are the same as the data type and attributes of the argument values. The result can be null.

The function is applied to the set of values derived from the argument values by the elimination of null values.

If the function is applied to an empty set, the result is the null value. Otherwise, the result is the minimum value in the set.

The specification of DISTINCT has no effect on the result and is not advised.

Example 1: Set the DECIMAL(15,2) variable *MIN_SALARY* to the minimum monthly salary of the employees represented in the sample table DSN8B10.EMP.

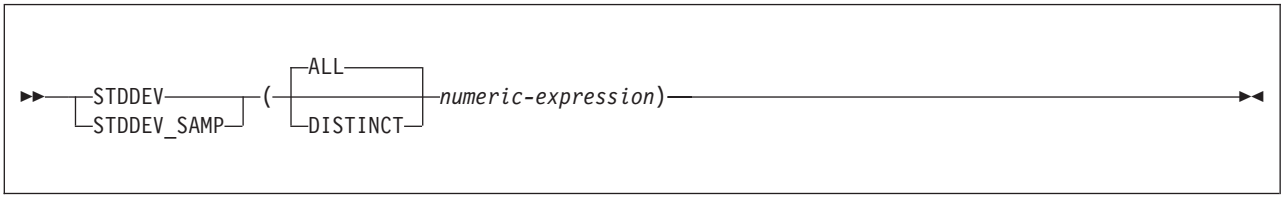
```
EXEC SQL SELECT MIN(SALARY) / 12
  INTO :MIN_SALARY
  FROM DSN8B10.EMP;
```

Example 2: Find the surname that comes first in the collating sequence for the employees represented in the sample table DSN8B10.EMP. Set the VARCHAR(15) variable *FIRST_NAME* to that surname.

```
EXEC SQL SELECT MIN(LASTNAME)
  INTO :FIRST_NAME
  FROM DSN8B10.EMP;
```

STDDEV or STDDEV_SAMP

The STDDEV or STDDEV_SAMP function returns the standard deviation (/*n*), or the sample standard deviation (/*n*-1), of a set of numbers.



The schema is SYSIBM.

The function returns the biased standard deviation (/*n*) or the sample standard deviation (/*n*-1) of a set of numbers, depending on which keyword is specified:

STDDEV
The formula that is used to calculate the biased standard deviation is logically equivalent to:

```
STDDEV = SQRT(VAR)
```

STDDEV_SAMP
The formula that is used to calculate the sample standard deviation is logically equivalent to:

```
STDDEV = SQRT(VARIANCE_SAMP)
```

The argument values must each be the value of any built-in numeric data type, and their sum must be within the range of the data type of the result.

The arguments can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34).

If the argument is DECFLOAT(*n*), the result of the function is DECFLOAT(34). Otherwise, the result of the function is double precision floating-point. The result can be null.

Before the function is applied to the set of values derived from the argument values, null values are eliminated. If DISTINCT is specified, redundant duplicate values are also eliminated.

If the function is applied to an empty set, the result is the null value. Otherwise, the result is the standard deviation of the values in the set.

The order in which the values are aggregated is undefined, but every intermediate result must be within the range of the result data type.

STDDEV_POP can be specified as a synonym for STDDEV.

Example: Using sample table DSN8B10.EMP, set the host variable *DEV*, which is defined as double precision floating-point, to the standard deviation of the salaries for the employees in department 'A00' (WORKDEPT='A00').

```
SELECT STDDEV(SALARY)
  INTO :DEV
  FROM DSN8B10.EMP
  WHERE WORKDEPT = 'A00';
```

For this example, host variable *DEV* is set to a double precision float-pointing number with an approximate value of '9742.43'.

SUM

The SUM function returns the sum of a set of numbers.

The schema is SYSIBM.

The argument values can be of any built-in numeric data type, and their sum must be within the range of the data type of the result.

The arguments can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34).

The data type of the result is determined as follows:

- DECFLOAT(34) if the argument is DECFLOAT(*n*).
- Large integer if the argument is small integer.
- Double precision floating-point if the argument is single precision floating-point.
- Otherwise, the result is the same as the data type of the argument.

The result can be null.

If the data type of the argument values is decimal, the scale of the result is the same as the scale of the argument values, and the precision of the result depends on the precision of the argument values and the decimal precision option:

- If the precision of the argument values is greater than 15 or the DEC31 option is in effect, the precision of the result is `min(31,`*P*`+10)`, where *P* is the precision of the argument values.
- Otherwise, the precision of the result is 15.

The function is applied to the set of values derived from the argument values by the elimination of null values. If DISTINCT is specified, redundant duplicate values are also eliminated.

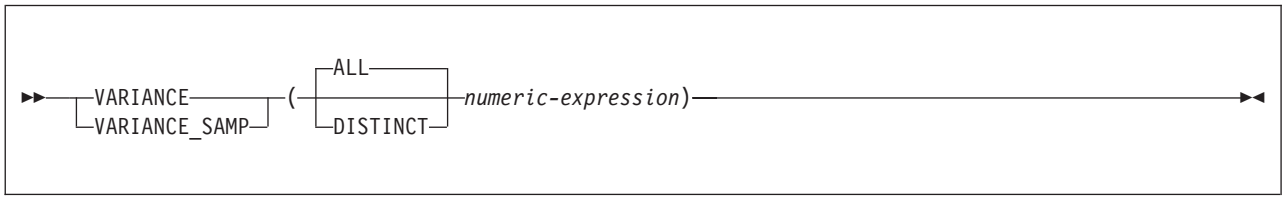
If the function is applied to an empty set, the result is the null value. Otherwise, the result is the sum of the values in the set. The order in which the summation is performed is undefined but every intermediate result must be within the range of the result data type.

Example: Set the large integer host variable *INCOME* to the total income from all sources (salaries, commissions, and bonuses) of the employees represented in the sample table DSN8B10.EMP. If DEC31 is not in effect, the resultant sum is DECIMAL(15,2) because all three columns are DECIMAL(9,2).

```
EXEC SQL SELECT SUM(SALARY+COMM+BONUS)
  INTO :INCOME
  FROM DSN8B10.EMP;
```

VARIANCE or VARIANCE_SAMP

The VARIANCE function returns the biased variance (/*n*) of a set of numbers. The VARIANCE_SAMP function returns the sample variance (/*n*-1) of a set of numbers.



The schema is SYSIBM.

The function returns the biased variance (/n) or the sample variance (/n-1) of a set of numbers, depending on which keyword is specified.

VARIANCE
The formula that is used to calculate the biased variance is logically equivalent to:

```
VARIANCE = SUM(X**2)/COUNT(X) - (SUM(X)/COUNT(X))**2
```

VARIANCE_SAMP
The formula that is used to calculate the sample variance is logically equivalent to:

```
VARIANCE_SAMP = (SUM(X**2) - ((SUM(X)**2) / (COUNT(*)))) / (COUNT(*) - 1)
```

The argument values can be of any built-in numeric type, and their sum must be within the range of the data type of the result. Before the function is applied to the set of values derived from the argument values, null values are eliminated. If DISTINCT is specified, redundant duplicate values are also eliminated.

The arguments can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34).

If the argument is DECFLOAT(*n*), the result of the function is DECFLOAT(34). Otherwise, the result of the function is double precision floating-point.

The result can be null; if any argument is null, the result is the null value.

Otherwise, the result is the variance of the values in the set.

The order in which the values are added is undefined, but every intermediate result must be within the range of the result data type.

Alternative syntax and synonyms:

- VAR or VAR_POP can be specified as synonym for VARIANCE
- VAR_SAMP can be specified as a synonym for VARIANCE_SAMP

Example 1: Using sample table DSN8B10.EMP, set host variable *VARNCE*, which is defined as double precision floating-point, to the variance of the salaries (SALARY) for those employees in department (WORKDEPT) 'A00'.

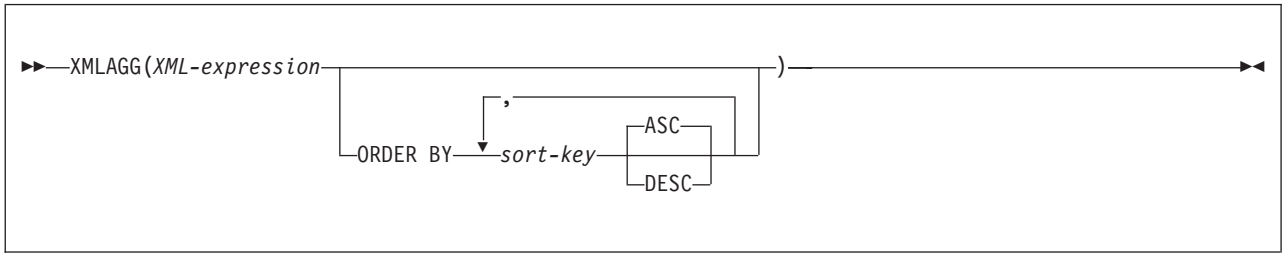
```
SELECT VARIANCE(SALARY)
  INTO :VARNCE
  FROM DSN8B10.EMP
  WHERE WORKDEPT = 'A00';
```

The result in *VARNCE* is set to a double precision-floating point number with an approximate value of '94915000.00'.

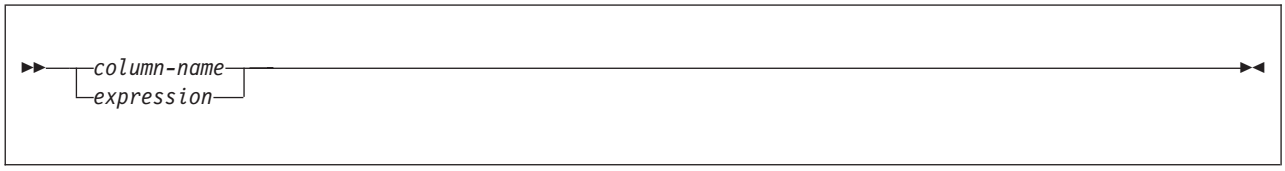
If VARIANCE_SAMP had been specified to find the sample variance of the salaries, the result in *VARNCE* would be set to a double precision-floating point number with an approximate value of '94915000.00'.

XMLAGG

The XMLAGG function returns an XML sequence that contains an item for each non-null value in a set of XML values.



sort-key



The schema is SYSIBM.

XML-expression
An expression that returns an XML value.

Unlike the arguments for other aggregate functions, a scalar fullselect is allowed in *XML-expression*.

ORDER BY
Specifies the order of the rows from the same grouping set that are processed in the aggregation. If the ORDER BY clause is not specified, or if the ORDER BY clause cannot differentiate the order of the sort key value, the rows in the same grouping set are arbitrarily ordered.

sort-key
Specifies a sort key value that is either a column name or an expression. The data type of the column or expression must not be a LOB or an XML value. A character string expression cannot have a length greater than 4000 bytes. If the sort key value is a constant, it does not refer to the position of the output column (as in the ordinary ORDER BY clause), but is simply a constant, which implies no sort key.

The ordering is based on the values of the sort keys, which might or might not be used in *XML-expression*.

If the sort key value is a character string that uses an encoding scheme other than Unicode, the ordering might be different. For example, a column PRODCODE uses EBCDIC. For two values, ('P001' and 'PA01'), relationship 'P001' > 'PA01' is true in EBCDIC, whereas 'P001' < 'PA01' is true in UTF-8. If the same sort key values are used in *XML-expression*, use the CAST specification to convert the sort key to Unicode to keep the ordering of XML values consistent with that of the sort key.

The function is applied to the set of values derived from the argument values by the elimination of null values.

The result can be null; if all *XML-expression* arguments are null. If the function is applied to an empty set, the result is the null value. Otherwise, the result is an XML sequence that contains an item for each value in the set.

Example: Group employees by their department, generate a 'Department' element for each department with its name as the attribute, nest all the 'emp' elements for employees in each department, and order the 'emp' elements by 'lname.'

```
SELECT XMLSERIALIZE(XMLDOCUMENT
       ( XMLELEMENT
       ( NAME "Department",
         XMLATTRIBUTES ( e.dept AS "name" ),
         XMLAGG ( XMLELEMENT ( NAME "emp", e.lname)
                  ORDER BY e.lname)
                ) ) AS "dept_list"
                  AS CLOB(1M))
  FROM employees e
  GROUP BY dept;
```

The result of the query would look similar to the following result:

```
   dept_list
 ---------------------------------------------
<Department name="Accounting">
  <emp>SMITH</emp>
  <emp>Yates</emp>
 </Department>
 <Department name="Shipping">
  <emp>Martin</emp>
  <emp>Oppenheimer</emp>
 </Department>
-----------------------------------------------
```

Scalar functions

A scalar function can be used wherever an expression can be used. The restrictions on the use of aggregate functions do not apply to scalar functions, because a scalar function is applied to single set of parameter values rather than to sets of values. The argument of a scalar function can be a function. However, the restrictions that apply to the use of expressions and aggregate functions also apply when an expression or aggregate function is used within a scalar function. For example, the argument of a scalar function can be a aggregate function only if a aggregate function is allowed in the context in which the scalar function is used.

If the argument of a scalar function is a string from a column with a field procedure, the function applies to the decoded form of the value and the result of the function does not inherit the field procedure.

Example: The following SELECT statement calls for the employee number, last name, and age of each employee in department D11 in the sample table DSN8B10.EMP. To obtain the ages, the scalar function YEAR is applied to the expression:

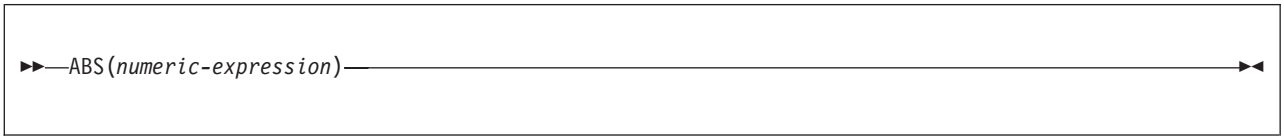
```
CURRENT DATE - BIRTHDATE
```

in each row of DSN8B10.EMP for which the employee represented is in department D11:

```
SELECT EMPNO, LASTNAME, YEAR(CURRENT DATE - BIRTHDATE)
  FROM DSN8B10.EMP
  WHERE WORKDEPT = 'D11';
```

ABS

The ABS function returns the absolute value of a number.



The schema is SYSIBM.

The argument must be an expression that returns a value of any built-in numeric data type.

The arguments can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34).

The result of the function has the same data type and length attribute as the argument.

The result can be null; if the argument is null, the result is the null value.

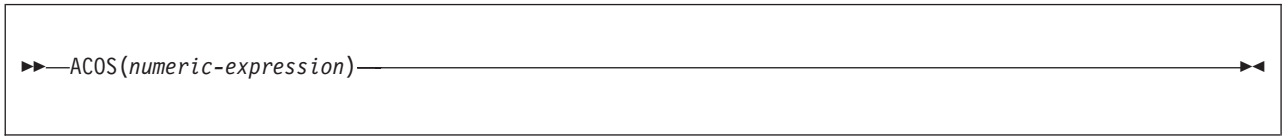
ABSVAL can be specified as a synonym for ABS. DB2 supports this keyword to provide compatibility with previous releases.

Example: Assume that host variable PROFIT is a large integer with a value of -50000. The following statement returns a large integer with a value of 50000.

```
SELECT ABS(:PROFIT)
  FROM SYSIBM.SYSDUMMY1;
```

ACOS

The ACOS function returns the arc cosine of the argument as an angle, expressed in radians. The ACOS and COS functions are inverse operations.



The schema is SYSIBM.

The argument must be an expression that returns the value of any built-in numeric data type except for DECFLOAT. The value must be greater than or equal to -1 and less than or equal to 1. If the argument is not a double precision floating-point number, it is converted to one for processing by the function.

The result of the function is a double precision floating-point number.

The result can be null; if the argument is null, the result is the null value.

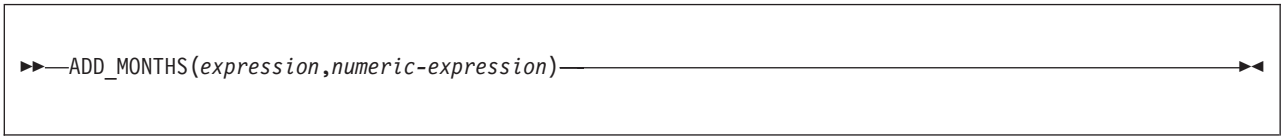
Example: Assume that host variable ACOSINE is DECIMAL(10,9) with a value of 0.070737202. The following statement:

```
SELECT ACOS(:ACOSINE)
  FROM SYSIBM.SYSDUMMY1;
```

returns a double precision floating-point number with an approximate value of 1.49.

ADD_MONTHS

The ADD_MONTHS function returns a date that represents *expression* plus a specified number of months.



The schema is SYSIBM.

expression
An expression that specifies the starting date. *expression* must return a value that is a date, timestamp, or a valid string representation of a date or timestamp. A string representation is a value that is a built-in character string data type or graphic string data type, that is not a LOB, and that has an actual length that is not greater than 255 bytes. A time zone in a string representation of a timestamp is ignored. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 101.

If *expression* is a TIMESTAMP WITH TIME ZONE value, *expression* is first cast to a TIMESTAMP WITHOUT TIME ZONE value with the same precision as *expression*. If *expression* is a string, *expression* is first cast to DATE.

numeric-expression
An expression that returns a value of any built-in numeric data type. The integer portion of *numeric-expression* specifies the number of months to add to the starting date specified by *expression*.A negative numeric value is allowed.

numeric-expression can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34).

If *expression* is a timestamp with a time zone value, or a valid string representation of a timestamp with a time zone value, the result is determined from the UTC representation of the datetime value.

If *expression* is a timestamp value the result is a TIMESTAMP WITHOUT TIME ZONE with the same precision as *expression*. Otherwise, the result is a DATE value.

The result can be null; if any argument is null, the result is the null value.

If *expression* is the last day of the month or if the resulting month has fewer days than the day component of *expression*, the result is the last day of the resulting month. Otherwise, the result has the same day component as *expression*. Any hours, minutes, seconds, or fractional seconds information included in *expression* is not changed by the function.

The result CCSID is the appropriate CCSID of the argument encoding scheme and the result subtype is the appropriate subtype of the CCSID.

Example 1: Assume today is January 31, 2007. Set the host variable ADD_MONTH with the last day of January plus 1 month.

```
SET :ADD_MONTH = ADD_MONTHS(LAST_DAY(CURRENT_DATE), 1);
```

The host variable ADD_MONTH is set with the value representing the end of February, 2007-02-28.

Example 2: Assume DATE is a host variable with the value July 27, 1965. Set the host variable ADD_MONTH with the value of that day plus 3 months.

```
SET :ADD_MONTH = ADD_MONTHS(:DATE,3);
```

The host variable ADD_MONTH is set with the value representing the day plus 3 months, 1965-10-27.

Example 3: It is possible to achieve similar results with the ADD_MONTHS function and date arithmetic. The following examples demonstrate the similarities and contrasts.

```
SET :DATEHV = DATE('2008-2-28') + 4 MONTHS;
SET :DATEHV = ADD_MONTHS('2008-2-28', 4);
```

In both cases, the host variable DATEHV is set with the value '2008–06–29'.

Now consider the same examples but with the date '2008–2–29' as the argument.

```
SET :DATEHV = DATE('2008-2-29') + 4 MONTHS;
```

The host variable DATEHV is set with the value '2008–06–29'.

```
SET :DATEHV = ADD_MONTHS('2008-2-29', 4);
```

The host variable DATEHV is set with the value '2008–06–30'.

In this case, the ADD_MONTHS function returns the last day of the month, which is June 30, 2008, instead of June 29, 2008. The reason is that February 29 is the last day of the month. So, the ADD_MONTHS function returns the last day of June.

Example 4: Assume *TSZ* is an SQL variable with the TIMESTAMP WITH TIME ZONE value 2008-02-29.20.00.00.000000-08.00. Set *TIMESZ* to the value of that TIMESTAMP WITH TIME ZONE plus 4 months. The string representation of the timestamp is first implicitly cast to TIMESTAMP WITHOUT TIME ZONE for the ADD_MONTHS function. The result of the ADD_MONTHS function does not contain a time zone.

```
SET TIMESZ: = ADD_MONTHS(TIMESTAMP_TZ(TSZ), 4);
```

With the string representation of a timestamp as input, the function returns a DATE value that represents the timestamp plus 4 months: 2008-06-30.

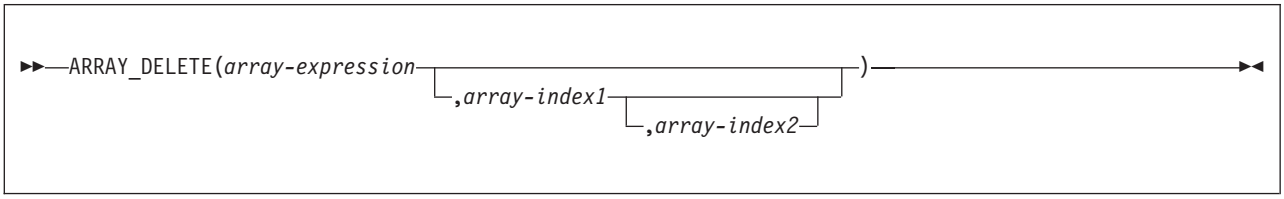
Example 5: Assume *TSZ* is a host variable with the value 2008-02-29.20.00.000000 -08.00 which is a string representation of a timestamp with a time zone. Set *TIMESZ* to the value of that timestamp with a time zone plus 4 months.

```
SET TIMESZ: = ADD_MONTHS(:TSZ, 4);
```

The host variable *TIMESZ* is set with the value that represents the timestamp with time zone plus 4 months, 2008-06-30-20.00.00.000000 -8.00.

ARRAY_DELETE

The ARRAY_DELETE function deletes elements from an array.



The schema is SYSIBM.

array-expression
An SQL variable or SQL parameter of an array type, or a CAST specification of a parameter marker to an array type.

array-index1
An expression that results in a value that is castable to the data type of the array index. If *array-expression* is an ordinary array, *array-index1* must be the null value.

array-index2
An expression that results in a value that is castable to the data type of the array index. If *array-expression* is an ordinary array, *array-index2* must be the null value. If *array-index2* is specified and is a non-null value, *array-index1* must be a non-null value that is less than the value of *array-index2*. If *array-index2* is the null value, ARRAY_DELETE is evaluated as if *array-index2* was not specified.

The result of ARRAY_DELETE has the same data type as *array-expression*.

If *array-index1* and *array-index2* are not specified, or they are the null value, all of the elements of *array-expression* are deleted, and the cardinality of the result array value is 0. If only *array-index1* is specified with a non-null value, the array element at index value *array-index1* is deleted. If *array-index2* is specified with a non-null value, the elements ranging from index value *array-index1* to *array-index2*, inclusive, are deleted.

The result can be null; if the first argument is null, the result is the null value.

The ARRAY_DELETE function can be invoked only in the following contexts:

- A source value for SET *assignment-statement*, an SQL PL *assignment-statement*, or a VALUES INTO statement
- The value that is returned in a RETURN statement in an SQL scalar function

Notes

Syntax alternatives: CAST (*SQL-variable* AS *array-type*) can be specified as an alternative to *SQL-variable*. CAST (*SQL-parameter* AS *array-type*) can be specified as an alternative to *SQL-parameter*.

Example 1: Suppose that ordinary array variable RECENT_CALLS has the array type PHONENUMBERS. Use ARRAY_DELETE to delete all the elements from RECENT_CALLS. Assign the result to the RECENT_CALLS array.

```
SET RECENT_CALLS = ARRAY_DELETE(RECENT_CALLS);
```

After the SET statement is executed, RECENT_CALLS is an empty array, which has a cardinality of zero.

An equivalent way of setting RECENT_CALLS to an empty array is to use an array constructor:

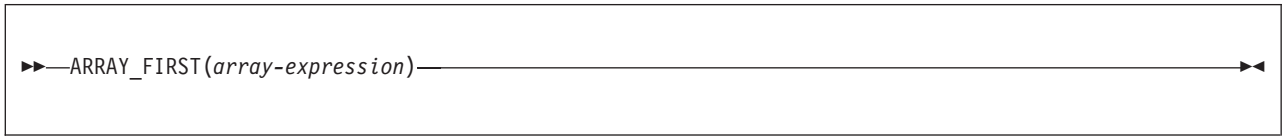
```
SET RECENT_CALLS = ARRAY[ ];
```

Example 2: Suppose that PRODUCTS is defined as an associative array type with VARCHAR values for the array index, and that variables FLOOR_TILES and REMAINIING_TILES are defined as arrays of the PRODUCTS array type. Use ARRAY_DELETE to assign the elements from the FLOOR_TILES array variable that do not have an index value between 'PK5100' and 'PS2500', inclusive, to the REMAINING_TILES array variable.

```
SET REMAINING_TILES = ARRAY_DELETE(FLOOR_TILES,'PK5100','PS2500');
```

ARRAY_FIRST

The ARRAY_FIRST function returns the minimum array index value of an array.



The schema is SYSIBM.

array-expression
An SQL variable or SQL parameter of an array type, or a CAST specification of a parameter marker to an array type.

The result of ARRAY_FIRST has the same data type as the array index. If *array-expression* is not null, and the array is not empty (the cardinality of the array is greater than 0), the value of the result is the minimum array index value, which is 1 for an ordinary array.

The result can be null; if the argument is null, the result is the null value.

If the array is empty (the cardinality of the array is 0), the result is the null value.

Notes

Syntax alternatives: CAST (*SQL-variable* AS *array-type*) can be specified as an alternative to *SQL-variable*. CAST (*SQL-parameter* AS *array-type*) can be specified as an alternative to *SQL-parameter*.

Example 1: Suppose that SPECIALNUMBERS is an ordinary array variable, and the elements of the array are integers. Return the first index value in the array variable SPECIALNUMBERS to the SQL variable E_CONSTIDX.

```
SET E_CONSTIDX = ARRAY_FIRST(SPECIALNUMBERS);
```

The result is 1.

Example 2: Suppose that PHONELIST is an associative array variable with VARCHAR index values. Values have been assigned to the elements in the array with the following statements:

```
SET PHONELIST['Home'] = '4443051234';
SET PHONELIST['Work'] = '4443052345';
SET PHONELIST['Cell'] = '4447893456';
```

The order in which values are assigned to array elements in an associative array does not matter. The elements of an associative array are stored in the array variable in ascending order of the associated array index values. After the values have been assigned to the PHONELIST array variable using the SET *assignment-statement* statements, the elements in the array variable are ordered as follows:

Index value	Element value
Cell	4447893456
Home	4443051234

Index value	Element value
Work	4443052345

Assign the value of the first index in the array variable to the character string variable named X.

```
SET X = ARRAY_FIRST(PHONELIST);
```

The value of 'Cell' is assigned to X because 'Cell' is the index value of the first element in the array variable.

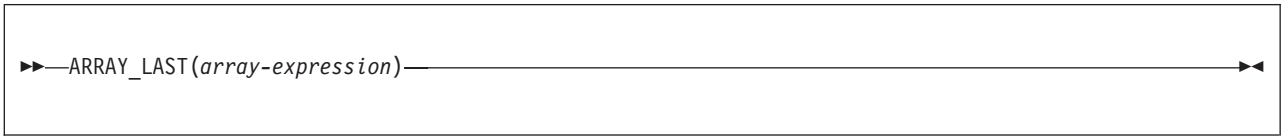
Assign the value of the array element with index X to the SQL variable NUMBER_TO_CALL.

```
SET NUMBER_TO_CALL = PHONELIST[X];
```

The assignment statement assigns the phone number '4447893456' to NUMBER_TO_CALL.

ARRAY_LAST

The ARRAY_LAST function returns the maximum array index value of an array.



The schema is SYSIBM.

array-expression
An SQL variable or SQL parameter of an array type, or a CAST specification of a parameter marker to an array type.

The result of ARRAY_LAST has the same data type as the array index, which is INTEGER for an ordinary array. If *array-expression* is not null, and the array is not empty (the cardinality of the array is greater than 0), the value of the result is the maximum array index value, which is the current cardinality of the array for an ordinary array.

The result can be null; if the argument is null, the result is the null value.

If the array is empty (the cardinality of the array is 0), the result is the null value.

Notes

Syntax alternatives: CAST (*SQL-variable* AS *array-type*) can be specified as an alternative to *SQL-variable*. CAST (*SQL-parameter* AS *array-type*) can be specified as an alternative to *SQL-parameter*.

Example 1: Suppose that SPECIALNUMBERS is an ordinary array variable, and the elements of the array are integers. The cardinality of the array is 10. Return the last index value in the array variable SPECIALNUMBERS to the SQL variable PI_CONSTIDX.

```
SET PI_CONSTIDX = ARRAY_LAST(SPECIALNUMBERS);
```

The result is 10.

Example 2: Suppose that PHONELIST is an associative array variable with VARCHAR index values. Values have been assigned to the elements in the array with the following statements:

```
SET PHONELIST['Home'] = '4443051234';
SET PHONELIST['Work'] = '4443052345';
SET PHONELIST['Cell'] = '4447893456';
```

The order in which values are assigned to array elements in an associative array does not matter. The elements of an associative array are stored in the array variable in ascending order of the associated array index values. After the values have been assigned to the PHONELIST array variable using the SET *assignment-statement* statements, the elements in the array variable are ordered as follows:

Index value	Element value
Cell	4447893456
Home	4443051234
Work	4443052345

Assign the value of the maximum index in the array variable to the character string variable named X.

```
SET X = ARRAY_LAST(PHONELIST);
```

The value of 'Work' is assigned to X because 'Work' is the index value of the last element in the array variable.

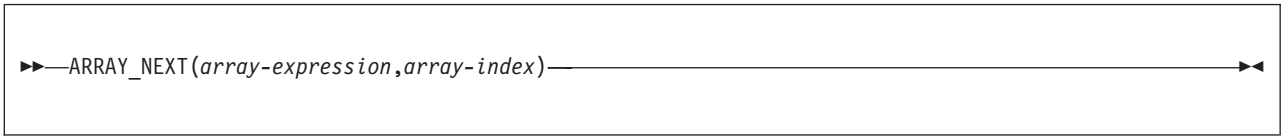
Assign the value of the array element with index X to the SQL variable NUMBER_TO_CALL.

```
SET NUMBER_TO_CALL = PHONELIST[X];
```

The assignment statement assigns the phone number '4443052345' to NUMBER_TO_CALL.

ARRAY_NEXT

The ARRAY_NEXT function returns the next larger array index value for an array, relative to a specified array index argument.



The schema is SYSIBM.

array-expression
An SQL variable or SQL parameter of an array type, or a CAST specification of a parameter marker to an array type.

array-index
An expression that results in a value that is castable to the data type of the array index. Valid values include any valid value for the data type.

array-index must *not* be an expression that references any of the following items:

- The CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP special register
- A nondeterministic function
- A function that is defined with EXTERNAL ACTION
- A function that is defined with MODIFIES SQL DATA
- A sequence expression

The result of ARRAY_NEXT is the next larger array index value defined in the array, relative to the specified *array-index* value. If *array-index* is less than the minimum index array value in the array, the result is the first array index value that is defined in the array.

The data type of the result has the same data type as the array index.

The result is null under the following conditions:

- *array-expression* or *array-index* is null
- The array that is represented by *array-expression* is empty (the cardinality of the array is 0)
- The value of *array-index* is greater than or equal to the value of the last index in the array

Notes

Syntax alternatives: CAST (*SQL-variable* AS *array-type*) can be specified as an alternative to *SQL-variable*. CAST (*SQL-parameter* AS *array-type*) can be specified as an alternative to *SQL-parameter*.

Example 1: Suppose that SPECIALNUMBERS is an ordinary array variable, and the elements of the array are integers. The cardinality of SPECIALNUMBERS is 10. Set the NEXT_CONSTIDX variable to the value of the array index for the SPECIALNUMBERS array element that follows the array element that is associated with an array index value of 9.

```
SET NEXT_CONSTIDX = ARRAY_NEXT(SPECIALNUMBERS,9);
```

The result is 10.

Example 2: Suppose that PHONELIST is an associative array variable with VARCHAR index values. Values have been assigned to the elements in the array with the following statements:

```
SET PHONELIST['Home'] = '4443051234';
SET PHONELIST['Work'] = '4443052345';
SET PHONELIST['Cell'] = '4447893456';
```

The order in which values are assigned to array elements in an associative array does not matter. The elements of an associative array are stored in the array variable in ascending order of the associated array index values. After the values have been assigned to the PHONELIST array variable using the SET *assignment-statement* statements, the elements in the array variable are ordered as follows:

Index value	**Element value**
Cell	4447893456
Home	4443051234
Work	4443052345

Assign the array index value that follows an array index value named 'Fax' to the character string variable named X.

```
SET X = ARRAY_NEXT(PHONELIST,'Fax');
```

Array index value 'Fax' does not exist, but the string 'Home' follows the string 'Fax' in sorting order. Therefore, 'Home' is assigned to X.

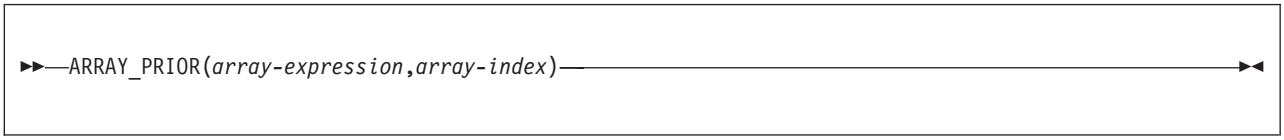
Assign the value of the array element with index X to the SQL variable NUMBER_TO_CALL.

```
SET NUMBER_TO_CALL = PHONELIST[X];
```

Because the value of X is 'Home', the assignment statement assigns the phone number '4443052345' to NUMBER_TO_CALL.

ARRAY_PRIOR

The ARRAY_PRIOR function returns the next smaller array index value for an array, relative to a specified array index argument.



The schema is SYSIBM.

array-expression
An SQL variable or SQL parameter of an array type, or a CAST specification of a parameter marker to an array type.

array-index
An expression that results in a value that is castable to the data type of the array index. Valid values include any valid value for the data type.

array-index must *not* be an expression that references any of the following items:

- The CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP special register
- A nondeterministic function
- A function that is defined with EXTERNAL ACTION
- A function that is defined with MODIFIES SQL DATA
- A sequence expression

The result of ARRAY_PRIOR is the next smaller array index value that is defined in the array, relative to the specified *array-index* value. If *array-index* is greater than the maximum index array value in the array, the result is the last array index value that is defined in the array.

The data type of the result has the same data type as the array index.

The result is null under the following conditions:

- *array-expression* or *array-index* is null.
- The array that is represented by *array-expression* is empty (the cardinality of the array is 0).
- The value of *array-index* is less than or equal to the value of the first index in the array.

Notes

Syntax alternatives: CAST (*SQL-variable* AS *array-type*) can be specified as an alternative to *SQL-variable*. CAST (*SQL-parameter* AS *array-type*) can be specified as an alternative to *SQL-parameter*.

Example 1: Suppose that SPECIALNUMBERS is an ordinary array variable, and the elements of the array are integers. The cardinality of SPECIALNUMBERS is 10. Set the PREV_CONSTIDX variable to the value of the array index for the SPECIALNUMBERS array element that precedes the array element that is associated with an array index value of 2.

```
SET PREV_CONSTIDX = ARRAY_PRIOR(SPECIALNUMBERS,2);
```

The result is 1.

Example 2: Suppose that PHONELIST is an associative array variable with VARCHAR index values. Values have been assigned to the elements in the array with the following statements:

```
SET PHONELIST['Home'] = '4443051234';
SET PHONELIST['Work'] = '4443052345';
SET PHONELIST['Cell'] = '4447893456';
```

The order in which values are assigned to array elements in an associative array does not matter. The elements of an associative array are stored in the array variable in ascending order of the associated array index values. After the values have been assigned to the PHONELIST array variable using the SET *assignment-statement* statements, the elements in the array variable are ordered as follows:

Index value	**Element value**
Cell	4447893456
Home	4443051234
Work	4443052345

Assign the array index value that precedes an array index value named 'Fax' to the character string variable named X.

```
SET X = ARRAY_PRIOR(PHONELIST,'Fax');
```

Array index value 'Fax' does not exist, but the string 'Cell' precedes the string 'Fax' in sorting order. Therefore, 'Cell' is assigned to X.

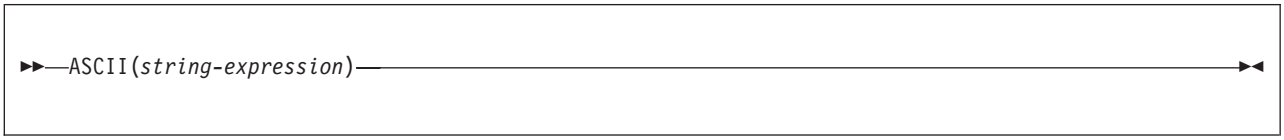
Assign the array index value that precedes array index value 'Cell' to the character string variable named X.

```
SET X = ARRAY_PRIOR(PHONELIST,'Cell');
```

The null value is assigned to X, because there is no array element before the array element with the index value 'Cell'.

ASCII

The ASCII function returns the leftmost character of the argument as an integer.



The schema is SYSIBM.

The argument can be any built-in character or graphic string data type, except for CLOB or DBCLOB. If the argument is an EBCDIC, Unicode, or graphic string, it is first converted to an SBCS ASCII character string (CCSID 367)[18] before the function is executed.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.

The result of the function is a large integer.

The result can be null; if the argument is null, the result is the null value.

Example: The following statement returns the ASCII value for the character 'A':

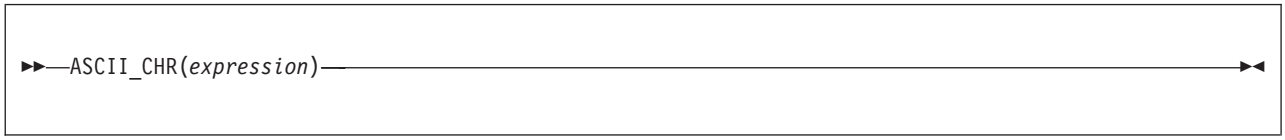
```
SET :hv = ASCII('A');
```

The host variable, :hv, is set to an integer with the value 65.

18. If the conversion does not exist, the ASCII function will return an error, or a substitution character might be returned.

ASCII_CHR

The ASCII_CHR function returns the character that has the ASCII code value that is specified by the argument.



The schema is SYSIBM.

expression
An expression that returns a built-in data type of BIGINT, INTEGER, or SMALLINT.

expression can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34) which is then assigned to a BIGINT value.

The result of the function is a fixed length character string encoded in the SBCS ASCII CCSID (regardless of the setting of the MIXED option in DSNHDECP). The length of the result is 1. If the value of *expression* is not in the range of 0 to 255, (0 to 127 if the SBCS ASCII CCSID for this system is CCSID 367) the null value is returned.

The result can be null; if the argument is null, the result is the null value.

CHR can be specified as a synonym for ASCII_CHR.

Example: Set :hv with the Euro symbol "€" in CCSID 923:

```
SET :hv = ASCII_CHR(164);  -- x'A4'
```

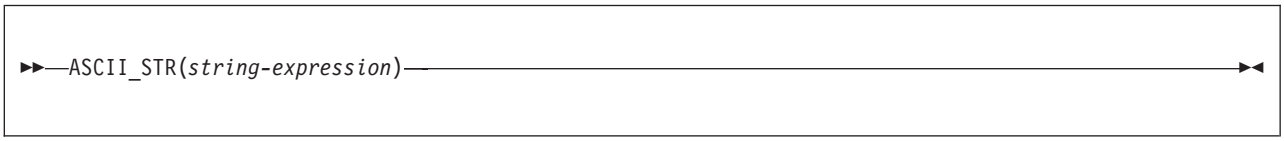
Set :hv with the Euro symbol "€" in CCSID 1252:

```
SET :hv = ASCII_CHR(128);  -- x'80'
```

In both cases, the "€" is assigned to :hv, but because the Euro symbol is located at different code points for the two CCSIDs, the input value is different.

ASCII_STR

The ASCII_STR function returns an ASCII version of the string in the system ASCII CCSID. The system ASCII CCSID is the SBCS ASCII CCSID on a MIXED=NO system or the MIXED ASCII CCSID on a MIXED=YES system.



The schema is SYSIBM.

string-expression
An expression that returns a value of a built-in character or graphic string. A character string must not be bit data. *string-expression* must be an ASCII, EBCDIC, or Unicode string.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.

ASCII_STR returns an ASCII version of the string. Non-ASCII characters are converted to UTF-16 characters and appear in the result in the form *xxxx* (or *xxxx**yyyy* for surrogate characters), where *xxxx* and *yyyy* represent a UTF-16 code unit.

The length attribute of the result will be `MIN( ( 5*n),32704)`. Where *n* is the result of applying the formulas in Table 30 on page 142 based on input and output data types.

The result of the function is a varying-length character string in the system ASCII CCSID. If the actual length of the result string exceeds the maximum for the return type, an error occurs.

The result can be null; if the argument is null, the result is the null value.

ASCIISTR can be specified as a synonym for ASCII_STR.

Example: The following example returns the ACSII string equivalent of the Unicode (UTF-8) string, '4869206D616D6520697320D090D0BDD0B4D180D0B5D0B9202020F0908080':

```
SET :HV1 =
ASCII_STR(X'4869206D616D6520697320D090D0BDD0B4D180D0B5D0B9202020F0908080');
```

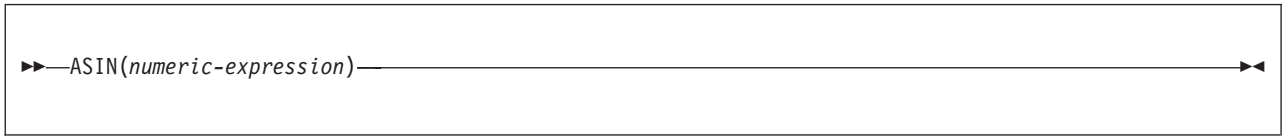
:HV1 is assigned the value 'Hi, my name is \0410\043D\0434\0440\0435\0439 \D800\DC00'. In this example, the UTF-8 characters D090, D0BD, D0B4, D180, D0B5, and D0B9 are converted to \0410\043D\0434\0440\0435\0439 and the non-ASCII character F0908080 is converted to \D800\DC00.

```
SET :HV1 = ASCII_STR('Hi, my name is Андрей (Andrei)');
```

:HV1 is assigned the value "Hi, my name is \0410\043D\0434\0440\0435\0439 (Andrei)"

ASIN

The ASIN function returns the arc sine of the argument as an angle, expressed in radians. The ASIN and SIN functions are inverse operations.



The schema is SYSIBM.

The argument must be an expression that returns the value of any built-in numeric data type except for DECFLOAT. The value must be greater than or equal to -1 and less than or equal to 1. If the argument is not a double precision floating-point number, it is converted to one for processing by the function.

The result of the function is a double precision floating-point number.

The result can be null; if the argument is null, the result is the null value.

The result is greater than or equal to $-\pi/2$ and less than or equal to $\pi/2$.

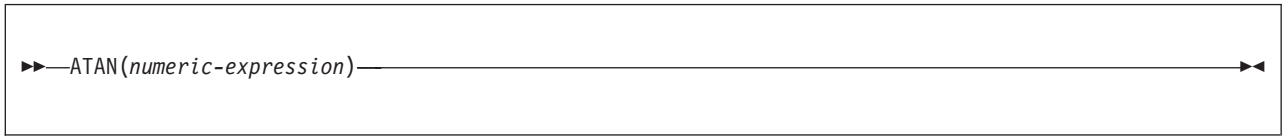
Example: Assume that host variable ASINE is DECIMAL(10,9) with a value of 0.997494987. The following statement:

```
SELECT ASIN(:ASINE)
  FROM SYSIBM.SYSDUMMY1;
```

returns a double precision floating-point number with an approximate value of 1.50.

ATAN

The ATAN function returns the arc tangent of the argument as an angle, expressed in radians. The ATAN and TAN functions are inverse operations.



The schema is SYSIBM.

The argument must be an expression that returns the value of any built-in numeric data type that is not DECFLOAT. The value must be greater than or equal to -1 and less than or equal to 1. If the argument is not a double precision floating-point number, it is converted to one for processing by the function.

The result of the function is a double precision floating-point number.

The result can be null; if the argument is null, the result is the null value.

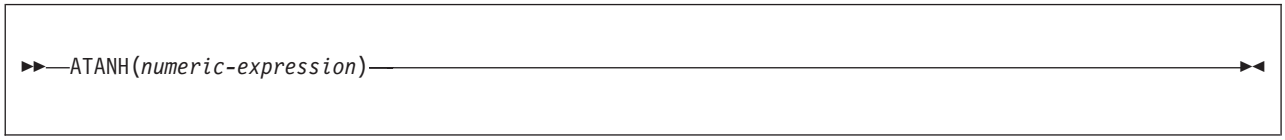
The result is greater than or equal to $-\pi/2$ and less than or equal to $\pi/2$.

Example: Assume that host variable ATANGENT is DECIMAL(10,9) with a value of 14.10141995. The following statement returns a double precision floating-point number with an approximate value of 1.50:

```
SELECT ATAN(:ATANGENT)
  FROM SYSIBM.SYSDUMMY1;
```

ATANH

The ATANH function returns the hyperbolic arc tangent of a number, expressed in radians. The ATANH and TANH functions are inverse operations.



The schema is SYSIBM.

The argument must be an expression that returns the value of any built-in numeric data type that is not DECFLOAT. The value must be greater than -1 and less than 1. If the argument is not a double precision floating-point number, it is converted to one for processing by the function.

The result of the function is a double precision floating-point number.

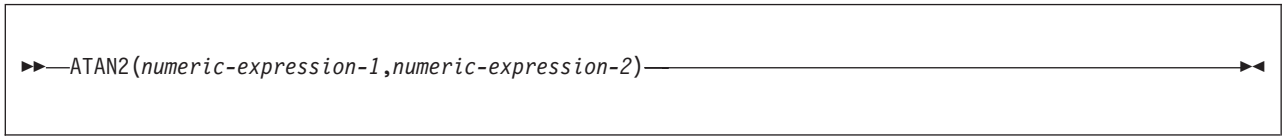
The result can be null; if the argument is null, the result is the null value.

Example: Assume that host variable HATAN is DECIMAL(10,9) with a value of 0.905148254. The following statement returns a double precision floating-point number with an approximate value of 1.50:

```
SELECT ATANH(:HATAN)
  FROM SYSIBM.SYSDUMMY1;
```

ATAN2

The ATAN2 function returns the arc tangent of *x* and *y* coordinates as an angle, expressed in radians.



The schema is SYSIBM.

The first and second arguments specify the *x* and *y* coordinates, respectively.

Each argument must be an expression that returns the value of any built-in numeric data type that is not DECFLOAT. Both arguments must not be 0. Any argument that is not a double precision floating-point number is converted to one for processing by the function.

The result of the function is a double precision floating-point number.

The result can be null; if any argument is null, the result is the null value.

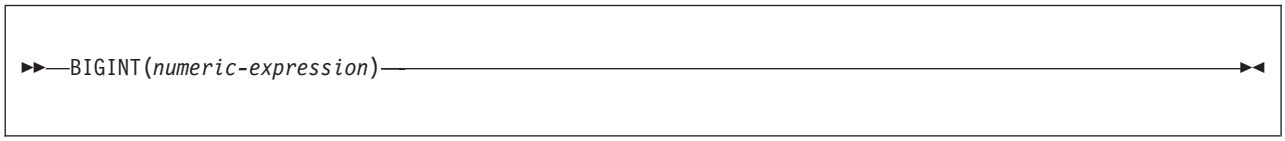
Example: Assume that host variables HATAN2A and HATAN2B are DOUBLE host variables with values of 1 and 2, respectively. The following statement returns a double precision floating-point number with an approximate value of 1.1071487:

```
SELECT ATAN2(:HATAN2A,:HATAN2B)
  FROM SYSIBM.SYSDUMMY1;
```

BIGINT

The BIGINT function returns a big integer representation of either a number or a character or graphic string representation of a number.

Numeric to Big Integer:



String to Big Integer:

The schema is SYSIBM.

Numeric to Big Integer

numeric-expression
An expression that returns a value of any built-in numeric data type.

The result is the same number that would occur if the argument were assigned to a big integer column or variable. If the whole part of the argument is not within the range of big integers, an error is returned. The fractional part of the argument is truncated.

String to Big Integer

string-expression
An expression that returns a value of a character or graphic string (except a CLOB and DBCLOB) with a length attribute that is not greater than 255 bytes. The string must contain a valid string representation of a number.

The result is the same number that would result from `CAST(`*string-expression*` AS BIGINT)`. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an integer constant. If the whole part of the argument is not within the range of big integers, an error is returned. Any fractional part of the argument is truncated.

The result of the function is a big integer.

The result can be null; if the argument is null, the result is the null value.

To increase the portability of applications, use the CAST specification.

Example 1: The following function returns the number 12345 (a BIGINT) for the number 12345.6:

```
SELECT BIGINT(12345.6)
   FROM SYSIBM.SYSDUMMY1;
```

Example 2: The following function returns a BIGINT value of 123456789012 for the number 00123456789012.

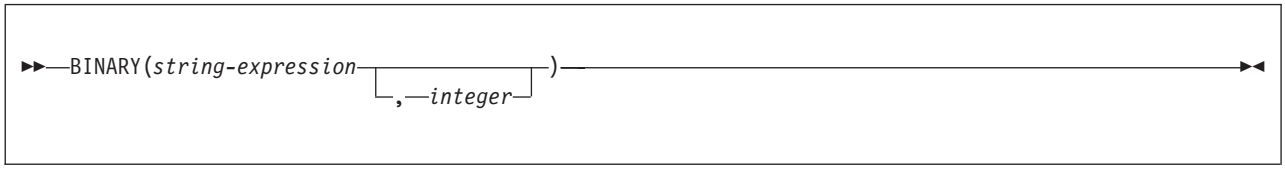
```
SELECT BIGINT('00123456789012')
   FROM SYSIBM.SYSDUMMY1;
```

Related reference:

"CAST specification" on page 267

BINARY

The BINARY function returns a BINARY (fixed-length binary string) representation of a string of any type or of a row ID type.



The schema is SYSIBM.

string-expression
An expression that returns a value that is a built-in character string, graphic string, binary string, or a row ID type.

integer
An integer value that specifies the length attribute of the resulting binary string. The value must be an integer between 1 and 255 inclusive.

If *integer* is not specified:

- If the *string-expression* is the empty string constant, an error occurs
- Otherwise, the length attribute of the result is the same as the length attribute of *string-expression*, except when the input is graphic data. In this case, the length attribute of the result is twice the length of *string-expression*.

The result of the function is a fixed-length binary string.

The result can be null; if the first argument is null, the result is the null value.

The actual length is the same as the length attribute of the result. If the length of the *string-expression* is less than the length of the result, the result is padded with hexadecimal zeroes up to the length of the result. If the length of the *string-expression* is greater than the length attribute of the result, truncation is performed. A warning is returned unless the first input argument is a character string and all the truncated characters are blanks, or the first input argument is a graphic string and all the truncated characters are double-byte blanks, or the first input argument is a binary string and all the truncated bytes are hexadecimal zeroes.

Following examples assume EBCDIC encoding of the input string constants.

Example 1: The following function returns a fixed-length binary string with a length attribute 1 and a value BX'00'.

```
SELECT BINARY('',1)
  FROM SYSIBM.SYSDUMMY1;
```

Example 2: The following function returns a fixed-length binary string with a length attribute 5 and a value BX'D2C2C80000'

```
SELECT BINARY('KBH',5)
  FROM SYSIBM.SYSDUMMY1;
```

Example 3: The following function returns a fixed-length binary string with a length attribute 3 and a value BX'D2C2C8'

```
SELECT BINARY('KBH')
  FROM SYSIBM.SYSDUMMY1;
```

Example 4: The following function returns a fixed-length binary string with a length attribute 3 and a value BX'D2C2C8'

```
SELECT BINARY('KBH ',3)
  FROM SYSIBM.SYSDUMMY1;
```

Example 5: The following function returns a fixed-length binary string with a length attribute 3 and a value BX'D2C2C8', a warning is also returned.

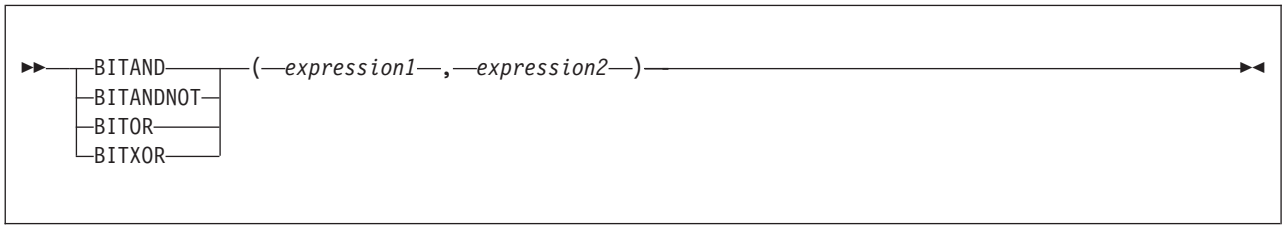
```
SELECT BINARY('KBH 93',3)
  FROM SYSIBM.SYSDUMMY1;
```

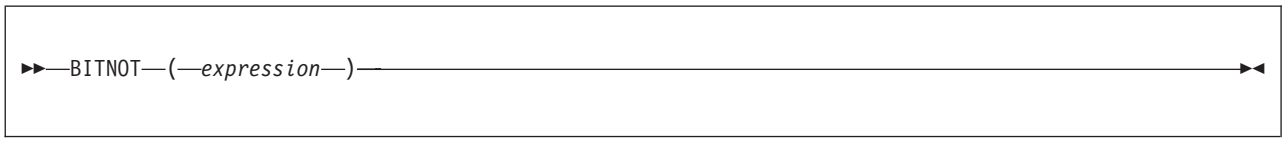
Example 6: The following function returns a fixed-length binary string with a length attribute 3 and a value BX'C1C2C3', a warning is also returned.

```
SELECT BINARY(BINARY('ABC',5),3)
  FROM SYSIBM.SYSDUMMY1;
```

BITAND, BITANDNOT, BITOR, BITXOR, and BITNOT

The bit manipulation functions operate on the *twos complement* representation of the integer value of the input arguments. The functions return the result as a corresponding base 10 integer value in a data type that is based on the data type of the input arguments.





The schema is SYSIBM.

Table 59. The bit manipulation funcitons

Function	Description	The bit in the *twos complement* representation of the result
BITAND	Performs a bitwise AND operation.	1 - only if the corresponding bits in both arguments are 1.
BITANDNOT	Clears any bit in the first argument that is in the second argument.	0 - if the corresponding bit in the second argument is 1. copied from the corresponding bit in the first argument - if the corresponding bit in the first argument is not 1.
BITOR	Performs a bitwise OR operation.	1 - unless the corresponding bits in both arguments are 0.
BITXOR	Performs a bitwise exclusive OR operation.	1 - unless the corresponding bits in both arguments are the same.
BITNOT	Performs a bitwise NOT operation.	Opposite of the corresponding bit in the argument.

expression, *expression1*, **or** *expression2*
: *expression*, *expression1*, or *expression2* must be integer values represented by the data types SMALLINT, INTEGER, BIGINT, or DECFLOAT. Arguments that are of type DECIMAL, REAL, or DOUBLE are cast to DECFLOAT. The value is truncated to a whole number.

The bit manipulation functions can operate on up to 16 bits for SMALLINT, 32 bits for INTEGER, 64 bits for BIGINT, and 113 bits for DECFLOAT. The range of supported DECFLOAT values includes integers from -2_{122} to 2^{122} - 1. Special values such as NaN or INFINITY are not supported.

If the two arguments have different data types, the argument that supports fewer bits is cast to a value with the data type of the argument that supports more bits. This cast impacts the bits that are set for negative values. For example, -1 as a SMALLINT value has 16 bits set to 1. When -1 is cast to an INTEGER value, it has 32 bits set to 1.

The result of the functions with two arguments has the data type of the argument that is highest in the data type precedence list for promotion. If either argument is DECFLOAT, the data type of the result is DECFLOAT(34).

The result of the BITNOT function has the same data type as the input argument, except that DECIMAL, REAL, DOUBLE, or DECFLOAT(16) returns DECFLOAT(34).

The result can be null; if any argument is null, the result is the null value.

Due to differences in internal representation between data types and on different hardware platforms, using functions (such as HEX) or host language constructs to view or compare internal representations of BIT function results and arguments is data type-dependent and not portable. The data type- and platform-independent way to view or compare BIT function results and arguments is to use the actual integer values.

The BITXOR function is can be used to toggle bits in a value.

The BITANDNOT function can be used to clear bits.

BITANDNOT(*val*, *pattern*) operates more efficiently than BITAND(*val*, BITNOT(*pattern*)).

The following examples are based on an ITEM table with a PROPERTIES column of type INTEGER.

Return all items for which the third property bit is set.

```
SELECT ITEMID FROM ITEM
  WHERE BITAND(PROPERTIES, 4) = 4;
```

Return all items for which the fourth or the sixth property bit is set.

```
SELECT ITEMID FROM ITEM
  WHERE BITAND(PROPERTIES, 40) <> 0;
```

Clear the twelfth property of the item whose ID is 3412.

```
UPDATE ITEM
  SET PROPERTIES = BITANDNOT(PROPERTIES, 2048)
  WHERE ITEMID = 3412;
```

Set the fifth property of the item whose ID is 3412.

```
UPDATE ITEM
  SET PROPERTIES = BITOR(PROPERTIES, 16)
  WHERE ITEMID = 3412;
```

Toggle the eleventh property of the item whose ID is 3412.

```
UPDATE ITEM
  SET PROPERTIES = BITXOR(PROPERTIES, 1024)
  WHERE ITEMID = 3412;
```

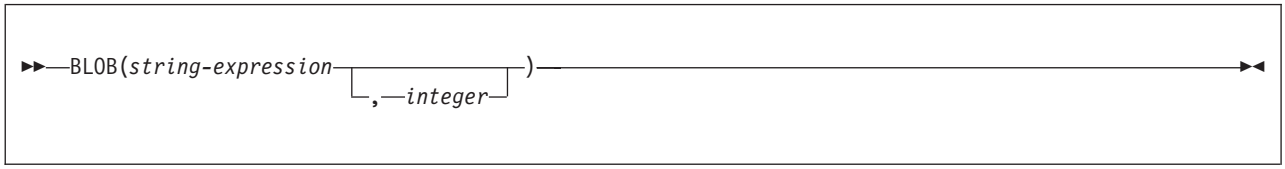
Switch all the bits in a 16-bit value that has only the second bit on.

```
SELECT  BITNOT(CAST(2 AS SMALLINT))
   FROM SYSIBM.SYSDUMMY1;
```

This example returns -3 (with a data type of SMALLINT).

BLOB

The BLOB function returns a BLOB representation of a string of any type or of a row ID type.



The schema is SYSIBM.

string-expression
An expression that returns a value that is a built-in character string, graphic string, binary string, or a row ID type.

integer
An integer value that specifies the length attribute of the resulting binary string. The value must be an integer between 1 and the maximum length of a BLOB.

Do not specify *integer* if *string-expression* is a row ID type.

If you do not specify *integer* and *string-expression* is an empty string constant, the length attribute of the result is 1, and the result is an empty string. Otherwise, the length attribute of the result is the same as the length attribute of *string-expression*, except when the input is graphic data. In this case, the length attribute of the result is twice the length of *expression*.

The result of the function is a BLOB.

The result can be null; if the first argument is null, the result is the null value.

The actual length of the result is the minimum of the length attribute of the result and the actual length of *string-expression* (or twice the length of *string-expression* when the input is graphic data). If the length of *string-expression* is greater than the length attribute of the result, truncation is performed. A warning is returned unless the first input argument is a character string and all the truncated characters are blanks, or the first input argument is a graphic string and all the truncated characters are double-byte blanks.

Example 1: The following function returns a BLOB for the string 'This is a BLOB'.

```
SELECT BLOB('This is a BLOB')
  FROM SYSIBM.SYSDUMMY1;
```

Example 2: The following function returns a BLOB for the large object that is identified by locator myclob_locator.

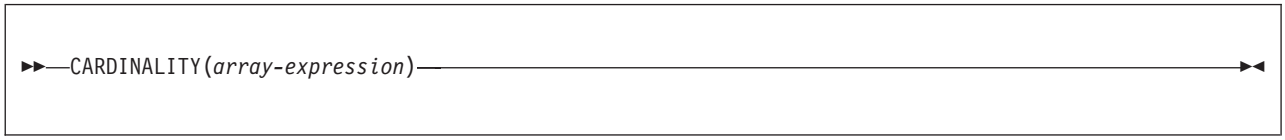
```
SELECT BLOB(:myclob_locator)
  FROM SYSIBM.SYSDUMMY1;
```

Example 3: Assume that a table has a BLOB column named TOPOGRAPHIC_MAP and a VARCHAR column named MAP_NAME. Locate any maps that contain the string 'Engles Island' and return a single binary string with the map name concatenated in front of the actual map.

```
SELECT BLOB(MAP_NAME || ':  ') || TOPOGRAPHIC_MAP
  FROM ONTARIO_SERIES_4
  WHERE TOPOGRAPHIC_MAP LIKE BLOB('%Engles Island%')
```

CARDINALITY

The CARDINALITY function returns a value of type BIGINT that represents the number of elements of an array.



The schema is SYSIBM.

array-expression
An SQL variable or SQL parameter of an array type, or a CAST specification of a parameter marker to an array type.

The result of the CARDINALITY function is as follows:

- For an ordinary array, the result is the highest array index for which the array has an assigned element. Elements that have been assigned the null value are considered to be assigned elements.
- For an associative array, the result is the actual number of unique array index values that are defined in *array-expression*.
- For an empty array, the result is 0.

The data type of the result is BIGINT.

The result is null if *array-expression* is null.

Notes

Syntax alternatives: CAST (*SQL-variable* AS *array-type*) can be specified as an alternative to *SQL-variable*. CAST (*SQL-parameter* AS *array-type*) can be specified as an alternative to *SQL-parameter*.

Example 1: Suppose that the array RECENT_CALLS is defined and contains a record of recent calls. RECENT_CALLS contains three elements. The following SET statement assigns the number of calls that have been stored in the array so far to SQL variable HOWMANYCALLS:

```
SET HOWMANYCALLS = CARDINALITY(RECENT_CALLS);
```

After the statement executes, HOWMANYCALLS contains 3.

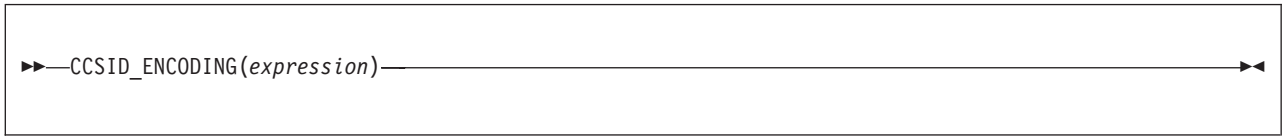
Example 2: Suppose that the associative array variable CANADACAPITALS of array type CAPITALSARRAY contains all of the capitals for the 10 provinces and three territories in Canada, as well as the capital of the country, Ottawa. The following SET statement assigns the cardinality of CANADACAPTITALS to SQL variable NUMCAPITALS.

```
SET NUMCAPITALS = CARDINALITY(CANADACAPITALS) ;
```

After the statement executes, CANADACAPITALS contains 14.

CCSID_ENCODING

The CCSID_ENCODING function returns a string value that indicates the encoding scheme of a CCSID that is specified by the argument.



The schema is SYSIBM.

expression

expression must be an expression that returns a value of a built-in numeric, character, or graphic string data type that is not a LOB. A character string must not have a length attribute greater than 255, and a graphic string must not have a length attribute greater than 127. If *expression* is a character or graphic string, the string must contain a valid string representation of a number. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming a numeric constant.

The function returns a value of ASCII, EBCDIC, UNICODE, or UNKNOWN depending on the CCSID specified by expression.

The result of the function is a fixed-length character string of length 8, which is padded on the right if necessary.

The result can be null; if the argument is null, the result is the null value.

The CCSID of the result is determined from the context in which the function was invoked. For more information, refer to "Determining the encoding scheme and CCSID of a string" on page 47.

Example 1: The following function returns a CCSID with a value for EBCDIC data.

```
SELECT CCSID_ENCODING(37) AS CCSID
  FROM SYSIBM.SYSDUMMY1;
```

Example 2: The following function returns a CCSID with a value for ASCII data.

```
SELECT CCSID_ENCODING(850) AS CCSID
  FROM SYSIBM.SYSDUMMY1;
```

Example 3: The following function returns a CCSID with a value for Unicode data.

```
SELECT CCSID_ENCODING(1208) AS CCSID
  FROM SYSIBM.SYSDUMMY1;
```

Example 4: The following function returns a CCSID with a value of UNKNOWN.

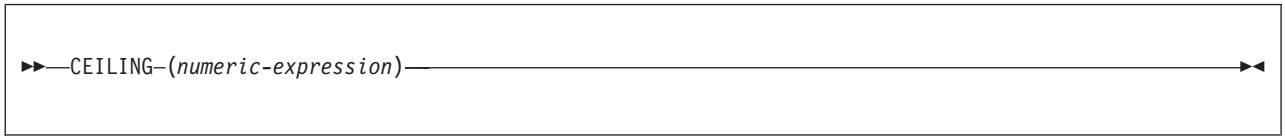
```
SELECT CCSID_ENCODING(1) AS CCSID
  FROM SYSIBM.SYSDUMMY1;
```

Example 5: The following function returns a CCSID with a value for EBCDIC data. The input data is a character string.

```
SELECT CCSID_ENCODING('37') AS CCSID
  FROM SYSIBM.SYSDUMMY1;
```

CEILING

The CEILING function returns the smallest integer value that is greater than or equal to the argument.



The schema is SYSIBM.

The argument must be an expression that returns a value of any built-in numeric data type.

The argument can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34).

The result of the function has the same data type and length attribute as the argument except that the scale is 0 if the argument is DECIMAL. For example, an argument with a data type of DECIMAL(5,5) results in DECIMAL(5,0).

The result can be null; if the argument is null, the result is the null value.

CEIL can be specified as a synonym for CEILING.

Example 1: The following statement shows the use of CEILING on positive and negative values:

```
SELECT CEILING(3.5), CEILING(3.1), CEILING(-3.1), CEILING(-3.5)
  FROM FROM SYSIBM.SYSDUMMY1;
```

This example returns: 04., 04., -03., -03.

Example 2: Using sample table DSN8B10.EMP, find the highest monthly salary for all the employees. Round the result up to the next integer. The SALARY column has a decimal data type.

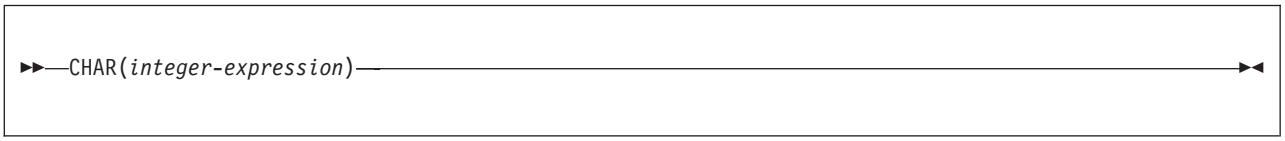
```
SELECT CEILING(MAX(SALARY)/12)
  FROM DSN8B10.EMP;
```

This example returns 04396. because the highest paid employee is Christine Haas who earns $52750.00 per year. Her average monthly salary before applying the CEILING function is 4395.83.

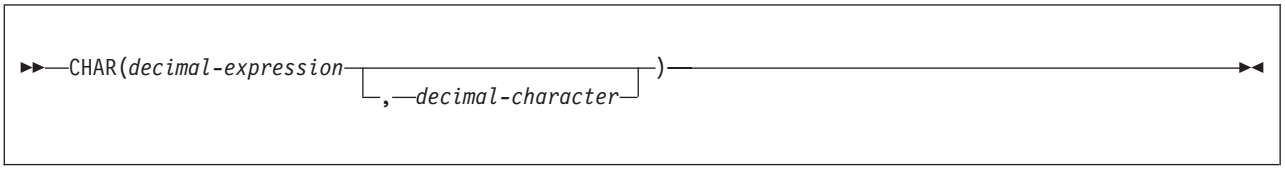
CHAR

The CHAR function returns a fixed-length character string representation of the argument.

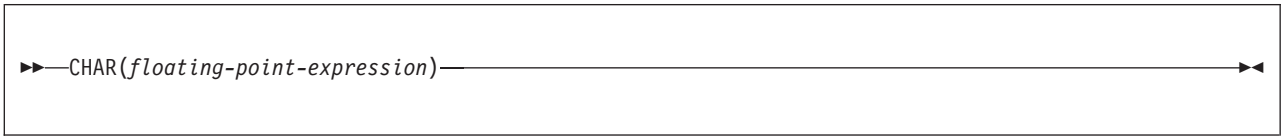
Integer to Character:



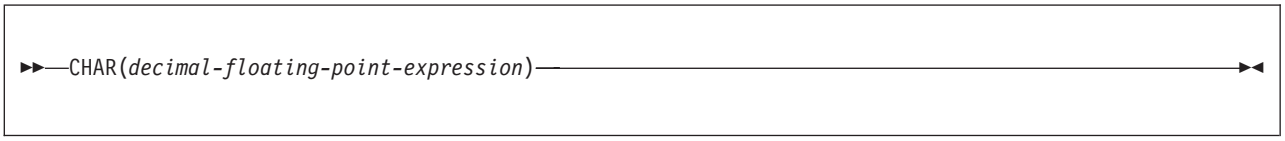
Decimal to Character:



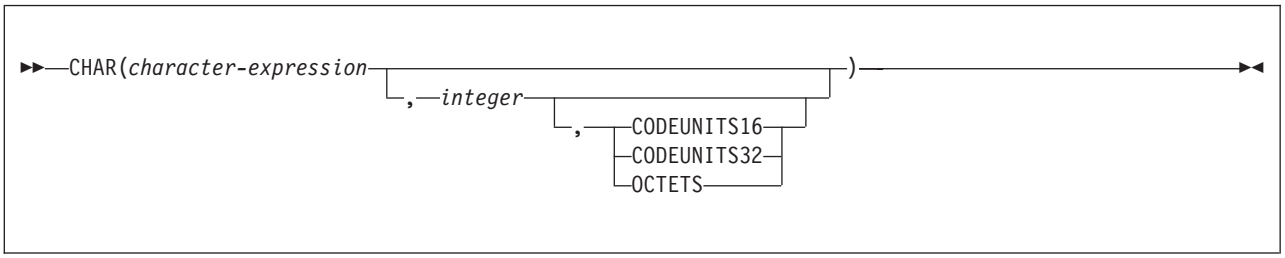
Floating-Point to Character:



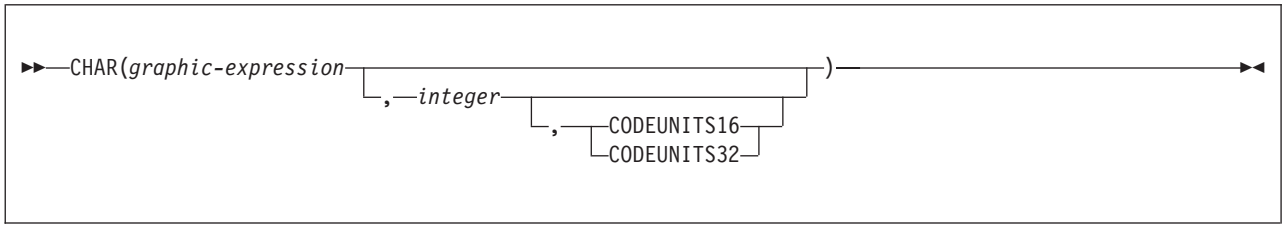
Decimal floating-point to Character:



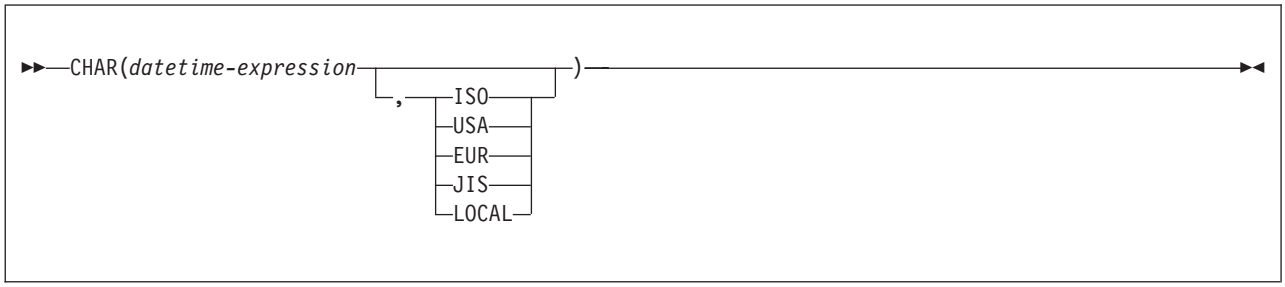
Character to Character:



Graphic to Character:



Datetime to Character:



Row ID to Character:

The schema is SYSIBM.

The CHAR function returns a fixed-length character string representation of one of the following values:

- An integer number if the first argument is a small, large, or big integer
- A decimal number if the first argument is a decimal number
- A floating-point number if the first argument is a single or double precision floating-point number
- A decimal floating-point number if the first argument is a decimal floating-point number
- A character string value if the first argument is any type of string
- A datetime value if the first argument is a date, time, or timestamp
- A row ID value if the first argument is a row ID

The result of the function is a fixed-length character string (CHAR).

The result can be null; if the first argument is null, the result is the null value.

Integer to Character

integer-expression
An expression that returns a value that is a built-in integer data type (SMALLINT, INTEGER, or BIGINT).

The result is the fixed-length character string representation of the argument in the form of an SQL integer constant. The result consists of *n* characters that are the significant digits that represent the value of the argument. If the argument is negative, the result has a preceding minus sign. The result is left justified, and its length depends on whether the argument is a small or large integer:

- For a small integer, the length of the result is 6. If the number of characters in the result is less than 6, the result is padded on the right with blanks to a length of 6.
- For a large integer, the length of the result is 11; if the number of characters in the result is less than 11, the result is padded on the right with blanks to a length of 11.

A positive value always includes one trailing blank.

The CCSID of the result is determined from the context in which the function is invoked. For more information, see "Determining the encoding scheme and CCSID of a string" on page 47.

Decimal to Character

decimal-expression
An expression that returns a value that is a built-in decimal data type. To specify a different precision and scale for the value of the expression, apply the DECIMAL function before applying the CHAR function.

decimal-character
Specifies the single-byte character constant (CHAR or VARCHAR) that is used to delimit the decimal digits in the result character string. The character must not be a digit, a plus sign (+), a minus sign (-), or a blank. The default is the period (.) or comma (,). For information on what factors govern the choice, see "Decimal point representation" on page 328.

The result is the fixed-length character string representation of the argument. The result includes a decimal character and up to *p* digits, where *p* is the precision of the *decimal-expression* with the preceding minus sign if the argument is negative. Leading zeros are not returned. Trailing zeros are returned. If the scale of *decimal-expression* is zero, the decimal character is not returned. If the number of bytes in the result is less than the defined length of the result, the result is padded on the right with blanks.[19]

If the function is invoked as CHAR9, the result is formatted as described in the previous paragraph with the following exceptions:

- The result includes leading zeros from the decimal value.
- A decimal character even if the scale of the decimal value is zero.
- A leading blank for a positive decimal value.

The leading blank is not returned for `CAST(`*`decimal-expression`*` AS CHAR(`*`n`*`))`.

The length of the result is2 +*p*, where *p* is the precision of the *decimal-expression*.

19. If the function is invoked as CHAR and the BIF_COMPATIBILITY subsystem parameter is set to V9, or if the function is invoked as V9_CURRENT.CHAR, the result is formatted the same as if the function was invoked as CHAR9.

The CCSID of the result is determined from the context in which the function was invoked. For more information, see "Determining the encoding scheme and CCSID of a string" on page 47.

Floating-Point to Character

floating-point-expression
An expression that returns a value that is a built-in floating-point data type (DOUBLE or REAL).

The result is the fixed-length character string representation of the argument in the form of an SQL floating-point constant. The length of the result is 24 bytes.

If the argument is negative, the first character of the result is a minus sign. Otherwise, the first character is a digit. If the value of the argument is zero, the result is 0E0. Otherwise, the result includes the smallest number of characters that can represent the value of the argument such that the mantissa consists of a single digit, other than zero, followed by a period and a sequence of digits.

If the number of characters in the result is less than 24, the result is padded on the right with blanks to length of 24.

The CCSID of the result is determined from the context in which the function is invoked. For more information, see "Determining the encoding scheme and CCSID of a string" on page 47.

Decimal floating-point to Character

decimal-floating-point-expression
An expression that returns a value that is a built-in decimal floating-point data type (DECFLOAT).

The result is the fixed-length character string representation of the argument in the form of an SQL decimal floating-point constant. The length of the result is 42 bytes. If the number of characters in the result is less than 42, the result is padded on the right with blanks to length of 42.

If the DECFLOAT value is one of the special values Infinity, sNaN, or NaN, the strings 'INFINITY', 'SNAN', or 'NAN', respectively, are returned. If the special value is negative, a minus sign is the first character in the returned string. The DECFLOAT special value sNaN does not result in an exception when it is converted to a string.

The CCSID of the result is determined from the context in which the function is invoked. For more information, see "Determining the encoding scheme and CCSID of a string" on page 47.

Character to Character

character-expression
An expression that returns a value of a built-in character string.

integer
The length attribute for the resulting fixed-length character string. The value must be an integer constant between 1 and 255.

If the length is not specified, the length attribute of the result is the minimum of 255 and the length attribute of *character-expression*. If *character-expression* is an empty string constant, an error occurs.

If **CODEUNITS16** or **CODEUNITS32** is specified, see "Determining the length attribute of the final result" on page 90 for information about how to calculate the length attribute of the result string.

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the unit that is used to express *integer*. If *character-expression* is a character string that is defined as bit data, **CODEUNITS16** and **CODEUNITS32** cannot be specified.

CODEUNITS16
Specifies that *integer* is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32
Specifies that *integer* is expressed in terms of 32-bit UTF-32 code units.

OCTETS
Specifies that *integer* is expressed in terms of bytes.

For more information about **CODEUNITS16**, **CODEUNITS32**, and **OCTETS**, see "String unit specifications" on page 87.

The actual length is the same as the length attribute of the result. If the length of *character-expression* is less than the length attribute of the result, the result is padded with blanks to the length of the result. If the length of *character-expression* is greater than the length attribute of the result, the result is truncated. Unless all of the truncated characters are blanks, a warning is returned.

If *character-expression* is bit data, the result is bit data. Otherwise, the CCSID of the result is the same as the CCSID of *character-expression*.

Graphic to Character

graphic-expression
An expression that returns a value of a built-in graphic string.

integer
The length attribute for the resulting fixed-length character string. The value must be an integer constant between 1 and 255.

If the length is not specified, the length attribute of the result is the minimum of 255 and the length attribute of *graphic-expression*. The length attribute of *graphic-expression* is (3 * *length*(*graphic-expression*)). If *graphic-expression* is an empty string constant, an error occurs.

If **CODEUNITS16** or **CODEUNITS32** is specified, see "Determining the length attribute of the final result" on page 90 for information about how to calculate the length attribute of the result string.

CODEUNITS16 or CODEUNITS32
Specifies the unit that is used to express *integer*.

CODEUNITS16
Specifies that *integer* is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32
Specifies that *integer* is expressed in terms of 32-bit UTF-32 code units.

For more information about **CODEUNITS16** and **CODEUNITS32**, see "String unit specifications" on page 87.

The actual length is the same as the length attribute of the result. If the length of *graphic-expression* is less than the length attribute of the result, the result is padded with blanks to the length of the result. If the length of *graphic-expression* is greater than the length attribute of the result, the result is truncated. Unless all of the truncated characters are blanks, a warning is returned.

The CCSID of the result is the character mixed CCSID that corresponds to the graphic CCSID of *graphic-expression*.

Datetime to Character

datetime-expression
An expression that is one of the following built-in data types:

date The result is the character string representation of the date in the format that is specified by the second argument. If the second argument is omitted, the DATE precompiler option, if one is provided, otherwise field DATE FORMAT on installation panel DSNTIP4 specifies the format. If the format is **LOCAL**, field LOCAL DATE LENGTH on installation panel DSNTIP4 specifies the length of the result. Otherwise, the length of the result is 10.

LOCAL denotes the local format at the DB2 subsystem that executes the SQL statement. If **LOCAL** is used for the format, a date exit routine must be installed at that DB2 subsystem.

An error occurs if the second argument is specified and is not a valid value.

time The result is the character string representation of the time in the format that is specified by the second argument. If the second argument is omitted, the TIME precompiler option, if one is provided, otherwise field TIME FORMAT on installation panel DSNTIP4 specifies the format. If the format is **LOCAL**, the field LOCAL TIME LENGTH on installation panel DSNTIP4 specifies the length of the result. Otherwise, the length of the result is 8.

LOCAL denotes the local format at the DB2 subsystem that executes the SQL statement. If **LOCAL** is used for the format, a time exit routine must be installed at that DB2 subsystem.

An error occurs if the second argument is specified and is not a valid value.

timestamp without time zone
The result is the character string representation of the timestamp. If *datetime-expression* is a TIMESTAMP(0) value, the length of the result is 19. If *datetime-expression* is a TIMESTAMP(*integer*) value, the length of the result is 20+*integer*. Otherwise, the length of the result is 26. The second argument must not be specified.

timestamp with time zone
The result is the character string representation of the timestamp with time zone, formatted as *yyyy-mm-dd-*

hh.mm.ss.nnnnnn±th:tm with the appropriate number of 'n' characters for the precision of the timestamp. If *datetime-expression* is a TIMESTAMP(0) WITH TIME ZONE, the length of the result is 147. If *datetime-expression* is a TIMESTAMP(*integer*) WITH TIME ZONE, the length of the result is 148+*integer*. The second argument must not be specified.

The CCSID of the result is determined from the context in which the function is invoked. For more information, see "Determining the encoding scheme and CCSID of a string" on page 47.

ISO, EUR, USA, JIS, or LOCAL
Specifies the date or time format of the resulting character string. For more information, see "String representations of datetime values" on page 101.

Row ID to Character

row-ID-expression
An expression that returns a value that is a built-in row ID data type.

The result is the fixed-length character string representation of the argument. The result is bit data.

The length of the result is 40. If the length of *row-ID-expression* is less than 40, the result is padded on the right with hexadecimal zeros to a length of 40.

Recommendation: To increase the portability of applications, use the CAST specification when the first argument is numeric, or the first argument is a string and the length argument is specified. For more information, see "CAST specification" on page 267.

Notes

Syntax alternatives: CHAR9 can be specified as an alternative to CHAR. The result of the the function is the same, except when the first argument is decimal data. See 'Decimal to Character' for a description of the result.

Example 1: HIREDATE is a DATE column in sample table DSN8B10.EMP. When it represents the date 15 December 1976 (as it does for employee 140), the following example returns the string value '12/15/1976' in character string variable *DATESTRING*:

```
EXEC SQL SELECT CHAR(HIREDATE, USA)
  INTO :DATESTRING
  FROM DSN8B10.EMP
  WHERE EMPNO = '000140';
```

Example 2: Host variable *HOUR* has a data type of DECIMAL(6,0) and contains a value of 50000. Interpreted as a time duration, this value is 5 hours. Assume that STARTING is a TIME column in some table. Then, when STARTING represents 17 hours, 30 minutes, and 12 seconds after midnight, the following example returns the value '10:30 PM':

```
CHAR(STARTING+:HOURS, USA)
```

Example 3: Assume that RECEIVED is defined as a TIMESTAMP column in table TABLEY. When the value of the date portion of RECEIVED represents the date 10

March 1997 and the time portion represents 6 hours and 15 seconds after midnight, the following example returns the string value '1997-03-10-06.00.15.000000':

```
SELECT CHAR(RECEIVED)
  FROM TABLEY
  WHERE INTCOL = 1234;
```

Example 4: For sample table DSN8B10.EMP, the following SQL statement sets the host variable *AVERAGE*, which is defined as CHAR(33), to the character string representation of the average employee salary.

```
EXEC SQL SELECT CHAR(AVG(SALARY))
  INTO :AVERAGE
  FROM DSN8B10.EMP;
```

With DEC31, the result of AVG applied to a decimal number is a decimal number with a precision of 31 digits. The only host languages in which such a large decimal variable can be defined are Assembler and C. For host languages that do not support such large decimal numbers, use the method shown in this example.

Example 5: For the rows in sample table DSN8B10.EMP, return the values in column LASTNAME, which is defined as VARCHAR(15), as a fixed-length character string and limit the length of the results to 10 characters.

```
SELECT CHAR(LASTNAME,10)
  FROM DSN8B10.EMP;
```

For rows that have a LASTNAME with a length greater than 10 characters (excluding trailing blanks), a warning that the value is truncated is returned.

Example 6: FIRSTNAME is a VARCHAR(12) column in a Unicode table T1. One of its values is the 6-character string 'Jürgen'. When FIRSTNAME has the values shown under 'Function', the results are shown under 'Returns':

```
Function ...                   Returns ...
-------------------------------------------------------------------------------
CHAR(FIRSTNAME,3,CODEUNITS32)  'Jür         ' -- x'4AC3BC722020202020202020'
CHAR(FIRSTNAME,3,CODEUNITS16)  'Jür      '    -- x'4AC3BC722020202020'
CHAR(FIRSTNAME,3,OCTETS)       'Jü'           -- x'4AC3BC'
```

Example 7: For the rows in sample table DSN8B10.EMP, return the values in column EDLEVEL, which is defined as SMALLINT, as a fixed-length character string.

```
SELECT CHAR(EDLEVEL)
  FROM DSN8B10.EMP;
```

An EDLEVEL of 18 is returned as CHAR(6) value '18 ' (18 followed by four blanks).

Example 8: In sample table DSN8B10.EMP, the SALARY column is defined as DECIMAL(9,2). For those employees who have a salary of 52750.00, return the hire date and the salary, using a comma as the decimal character in the salary (52750,00).

```
SELECT HIREDATE, CHAR(SALARY, ',')
  FROM DSN8B10.EMP
  WHERE SALARY = 52750.00;
```

The salary is returned as the string value '52750,00'.

Example 9: Repeat the scenario in Example 8 except subtract the SALARY column from 60000.00 and return the salary with the default decimal character.

```
SELECT HIREDATE, CHAR (60000.00 - SALARY)
  FROM DSN8B10.EMP
  WHERE SALARY = 52750.00;
```

The salary is returned as the string value '7250.00'.

Example 10: Assume that host variable *SEASONS_TICKETS* is defined as INTEGER and has a value of 10000. Use the DECIMAL and CHAR functions to change the value into the character string ' 10000.00'.

```
SELECT CHAR(DECIMAL(:SEASONS_TICKETS,7,2))
  FROM SYSIBM.SYSDUMMY1;
```

Example 11: Assume that columns COL1 and COL2 in table T1 are both defined as REAL and that T1 contains a single row with the values 7.1E+1 and 7.2E+2 for the two columns. Add the two columns and represent the result as a character string.

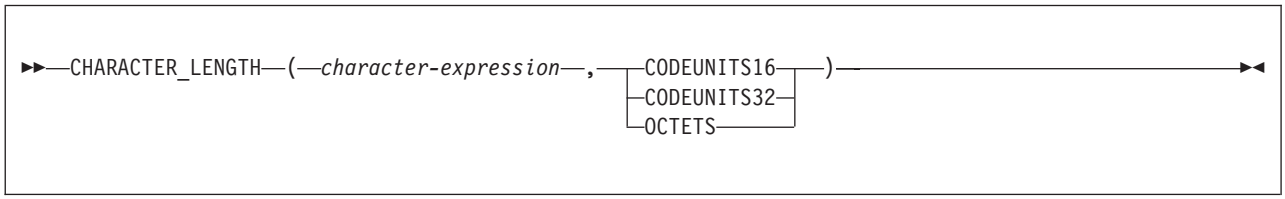
```
SELECT CHAR(COL1 + COL2)
  FROM T1;
```

The result is the character value '1.43E2 '.

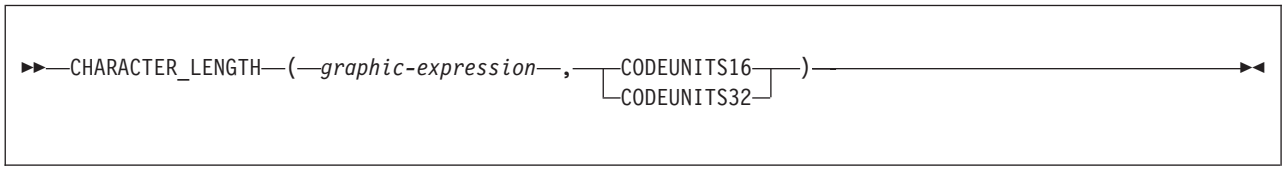
CHARACTER_LENGTH

The CHARACTER_LENGTH function returns the length of the first argument in the specified string unit.

Character string:



Graphic string:



The schema is SYSIBM.

Character string:

character-expression
An expression that returns a value of a built-in character string. *character-expression* cannot be bit data.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the unit that is used to express the length of the result.

CODEUNITS16
Specifies that the result is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32
Specifies that the result is expressed in terms of 32-bit UTF-32 code units.

OCTETS
Specifies the result is expressed in terms of bytes.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String unit specifications" on page 87.

The result of the function is a large integer.

The result can be null; if the argument is null, the result is the null value.

The result is the length of *character-expression* expressed in the number of string units that were specified. The length of fixed-length strings includes trailing blanks. The length of varying-length strings is the actual length and not the maximum length.

Graphic string:

graphic-expression
An expression that returns a value of a built-in graphic string.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.

CODEUNITS16 or CODEUNITS32
Specifies the unit that is used to express the length of the result.

CODEUNITS16
Specifies that the result is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32
Specifies that the result is expressed in terms of 32-bit UTF-32 code units.

For more information about CODEUNITS16 and CODEUNITS32 see "String unit specifications" on page 87.

The result of the function is a large integer.

The result can be null; if the argument is null, the result is the null value.

The result is the length of *graphic-expression* expressed in the number of string units that were specified. The length of fixed-length strings includes trailing blanks. The length of varying-length strings is the actual length and not the maximum length.

Example: Assume that NAME is a VARCHAR(128) column, encoded in Unicode UTF-8, that contains the value 'Jürgen'. The following two queries return the value 6:

```
SELECT CHARACTER_LENGTH(NAME,CODEUNITS32)
  FROM T1 WHERE NAME = 'Jürgen';
SELECT CHARACTER_LENGTH(NAME,CODEUNITS16)
  FROM T1 WHERE NAME = 'Jürgen';
```

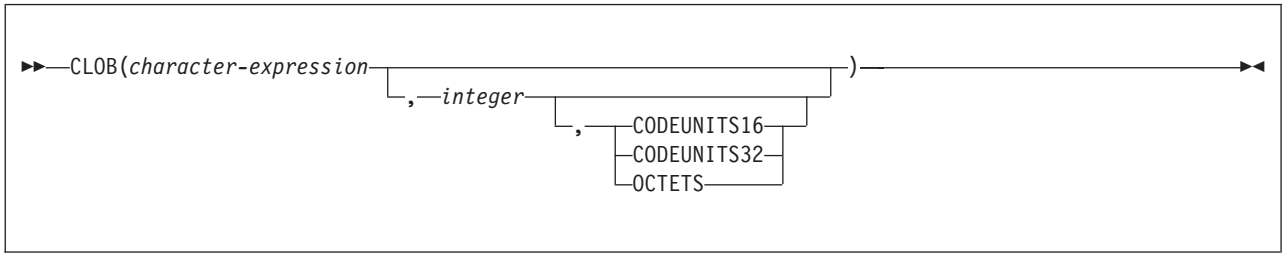
The following two queries return the value 7:

```
SELECT CHARACTER_LENGTH(NAME,OCTETS)
  FROM T1 WHERE NAME = 'Jürgen';
SELECT LENGTH(NAME)
  FROM T1 WHERE NAME = 'Jürgen';
```

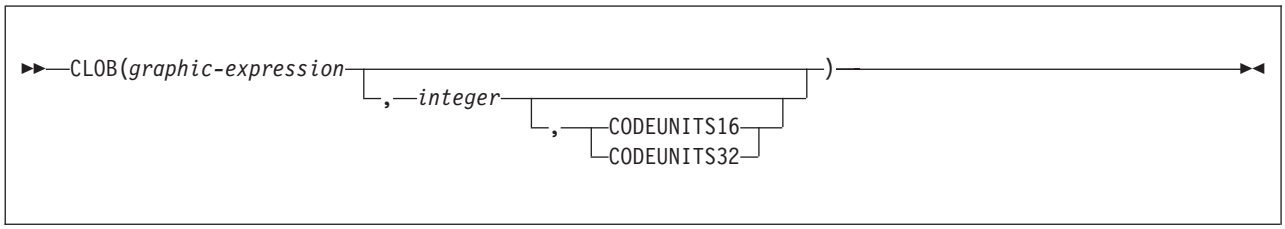
CLOB

The CLOB function returns a CLOB representation of a string.

Character to CLOB:



Graphic to CLOB:



The schema is SYSIBM.

Character to CLOB

character-expression
An expression that returns a value of a character string. If *character-expression* is bit data, an error occurs.

integer
An integer constant that specifies the length attribute of the resulting CLOB data type. The value must be between 1 and the maximum length of a CLOB, expressed in the units that are either implicitly or explicitly specified.

If you do not specify *integer* and *character-expression* is an empty string constant, the length attribute of the result is 1, and the result is an empty string. Otherwise, the length attribute of the result is the same as the length attribute of *character-expression*.

If CODEUNITS16 or CODEUNITS32 is specified, see "Determining the length attribute of the final result" on page 90 for information on how to calculate the length attribute of the result string.

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the unit that is used to express *integer*.

CODEUNITS16
Specifies that *integer* is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32
Specifies that *integer* is expressed in terms of 32-bit UTF-32 code units.

OCTETS
Specifies that *integer* is expressed in terms of bytes.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String unit specifications" on page 87.

The result of the function is a CLOB.

The result can be null; if the first argument is null, the result is the null value.

The actual length of the result is the minimum of the length attribute of the result and the actual length of *character-expression*. If the length of *character-expression* is greater than the length specified, the result is truncated. Unless all of the truncated characters are blanks, a warning is returned.

The CCSID of the result is the same as the CCSID of *character-expression*.

Graphic to CLOB

graphic-expression
An expression that returns a value of a graphic string.

integer
An integer constant that specifies the length attribute of the resulting CLOB data type. The value must be between 1 and the maximum length of a CLOB, expressed in the units that are either implicitly or explicitly specified.

If you do not specify *integer* and *graphic-expression* is an empty string constant, the length attribute of the result is 1, and the result is an empty string. Otherwise, the length attribute of the result is (3 * *length*(*graphic-expression*)).

If CODEUNITS16 or CODEUNITS32 is specified, see "Determining the length attribute of the final result" on page 90 for information on how to calculate the length attribute of the result string.

CODEUNITS16 or CODEUNITS32
Specifies the unit that is used to express *integer*.

CODEUNITS16
Specifies that *integer* is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32
Specifies that *integer* is expressed in terms of 32-bit UTF-32 code units.

For more information about CODEUNITS16 and CODEUNITS32, see "String unit specifications" on page 87.

The result of the function is a CLOB.

The result can be null; if the first argument is null, the result is the null value.

The actual length of the result is the minimum of the length attribute of the result and the actual length of *graphic-expression*. If the length of *graphic-expression* is greater than the length specified, the result is truncated. Unless all of the truncated characters are blanks, a warning is returned.

The CCSID of the result is the character mixed CCSID that corresponds to the graphic CCSID of *graphic-expression*.

Example 1: The following function returns a CLOB for the string 'This is a CLOB'.

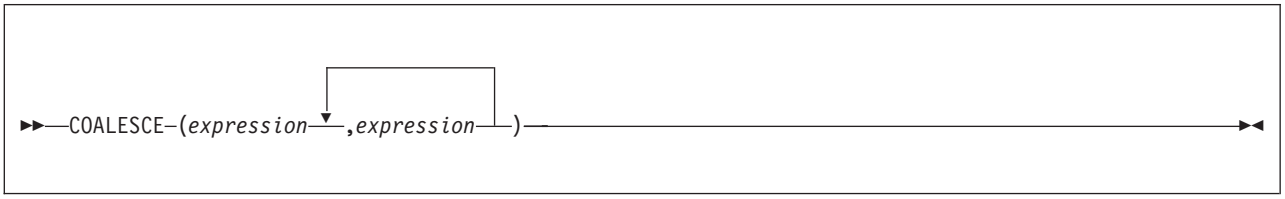
```
SELECT CLOB('This is a CLOB')
  FROM SYSIBM.SYSDUMMY1;
```

Example 2: FIRSTNME is a VARCHAR(12) column in table T1. One of its values is the 6-character string 'Jürgen'. When FIRSTNME has this value:

```
Function ...                        Returns ...
CLOB(FIRSTNME,3,CODEUNITS32)        'Jür'  -- x'4AC3BC72'
CLOB(FIRSTNME,3,CODEUNITS16)        'Jür'  -- x'4AC3BC72'
CLOB(FIRSTNME,3,OCTETS)             'Jü'   -- x'4AC3BC'
```

COALESCE

The COALESCE function returns the value of the first nonnull expression.



The schema is SYSIBM.

The arguments must be compatible. For more information on compatibility, refer to the compatibility matrix in Table 23 on page 121. The arguments can be of either a built-in or user-defined data type. The COALESCE function allows multiple arrays with the same user-defined array type, but does not allow a mixture of array variables that have a user-defined array type and array values that do not have a user-defined array type.

The COALESCE function cannot be used as a source function when creating a user-defined function.

The arguments are evaluated in the order in which they are specified, and the result of the function is the first argument that is not null. The result can be null only if all arguments can be null. The result is null only if all arguments are null.

The selected argument is converted, if necessary, to the attributes of the result. The attributes of the result are determined using the "Rules for result data types" on page 144. If the COALESCE function has more than two arguments, the rules are applied to the first two arguments to determine a candidate result type. The rules are then applied to that candidate result type and the third argument to determine another candidate result type. This process continues until all arguments are analyzed and the final result type is determined.

If there are any mixed character string or graphic string and numeric arguments, the string value is implicitly cast to a DECFLOAT(34) value.

The COALESCE function can also handle a subset of the functions provided by CASE expressions. The result of using `COALESCE(e1,e2)` is the same as using the expression:

```
CASE WHEN e1 IS NOT NULL THEN e1 ELSE e2 END
```

VALUE can be specified as a synonym for COALESCE.

Example 1: Assume that SCORE1 and SCORE2 are SMALLINT columns in table GRADES, and that nulls are allowed in SCORE1 but not in SCORE2. Select all the rows in GRADES for which SCORE1 + SCORE2 > 100, assuming a value of 0 for SCORE1 when SCORE1 is null.

```
SELECT * FROM GRADES
  WHERE COALESCE(SCORE1,0) + SCORE2 > 100;
```

Example 2: Assume that a table named DSN8B10.EMP contains a DATE column named HIREDATE, and that nulls are allowed for this column. The following

query selects all rows in DSN8B10.EMP for which the date in HIREDATE is either unknown (null) or earlier than 1 January 1960.

```
SELECT * FROM DSN8B10.EMP
  WHERE COALESCE(HIREDATE,DATE('1959-12-31')) < '1960-01-01';
```

The predicate could also be coded as `COALESCE(HIREDATE,'1959-12-31')` because, for comparison purposes, a string representation of a date can be compared to a date.

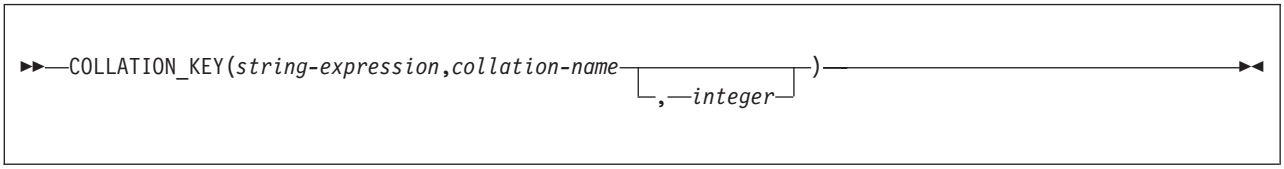
Example 3: Assume that for the years 1993 and 1994 there is a table that records the sales results of each department. Each table, S1993 and S1994, consists of a DEPTNO column and a SALES column, neither of which can be null. The following query provides the sales information for both years.

```
SELECT COALESCE(S1993.DEPTNO,S1994.DEPTNO) AS DEPT, S1993.SALES, S1994.SALES
  FROM S1993 FULL JOIN S1994 ON S1993.DEPTNO = S1994.DEPTNO
  ORDER BY DEPT;
```

The full outer join ensures that the results include all departments, regardless of whether they had sales or existed in both years. The COALESCE function allows the two join columns to be combined into a single column, which enables the results to be ordered.

COLLATION_KEY

The COLLATION_KEY function returns a varying-length binary string that represents the collation key of the argument in the specified collation.



The schema is SYSIBM.

The result of COLLATION_KEY on one string can be compared in binary form with the result of COLLATION_KEY on another string to determine their order within the specified *collation-name*. For the comparison to be meaningful, the results of the COLLATION_KEY must be from the same *collation-name*.

string-expression
An expression that returns a character or graphic string that is not a LOB for which the collation key is to be determined. If *string-expression* is a character string, it must not be FOR BIT DATA. If *string-expression* is not in Unicode UTF-16 (CCSID 1200), it is converted to Unicode UTF-16 before the corresponding collation key is obtained. The length of *string-expression* must not exceed 32704 bytes of the UTF-16 representation.

collation-name
A string constant or a string host variable that is not a binary string, CLOB, or DBCLOB. *collation-name* specifies the collation to use when determining the collation key. If *collation-name* is not an EBCDIC value, it is converted to EBCDIC. The length of *collation-name* must be between 1 and 255 bytes of the EBCDIC representation. The value of *collation-name* is not case sensitive and must be a left justified, valid "short path" collation setting for the parameter CUNBOPRM_Collation_Keyword in area CUN4BOPR. For detailed information about the "short path" setting in the parameter CUNBOPRM_Collation_Keyword, see *z/OS Support for Unicode: Using Conversion Services*.

The value of the host variable must not be null. If the host variable has an associated indicator variable, the value of the indicator variable must not indicate a null value. *collation-name* must be left justified within the host variable. It must also be padded on the right with blanks if the length is less than that of the host variable and the host variable is a fixed length CHAR or GRAPHIC data type.

collation-name is in the form of CUN4BOPR_Collation_Keyword specification. You must specify a value that is acceptable for the z/OS CUNBOPR_Collation_Keyword parameter.

The following table lists some supported values:

Table 60. Collation Keywords Reference

Attribute name	Key	Possible values
Locale	L.R.V	<locale>
Strength	S	1, 2, 3, 4, I, D
Case_Level	K	X, O, D

Table 60. Collation Keywords Reference (continued)

Attribute name	Key	Possible values
Case_First	C	X, L, U, D
Alternate	A	N, S, D
Variable_Top	T	<hex digits>
Normalization	N	X, O, D
French	F	X, O, D
Hinayana	H	X, O, D

The following table describes the abbreviations for the collation keywords:

Abbreviation	Definition
D	default
O	on
X	off
1	primary
2	secondary
3	tertiary
4	quaternary
I	identical
S	shifted
N	non-ignorable
L	lower-first
U	upper-first

The following examples show keywords using the above specifications:

```
'UCA400R1_AS_LSV_S3_CU'
```

UCA version 4.0.1; ignore spaces, punctuation and symbols; use Swedish linguistic conventions; use case-first upper; compare case-sensitive.

```
'UCA400R1_AN_LSV_S3_CL_NO'
```

UCA version 4.0.1; do not ignore spaces, punctuation and symbols; use Swedish linguistic conventions; use case-first lower (or does not set it to mean the same, since lower is used in most locales as the default); normalization ON; compare case-sensitive.

integer

An integer value that specifies the length attribute of the result. If specified, the value must be an integer constant between 1 and 32704.

If the length is not specified, the length attribute of the result is determined as follows:

string-expression	**Result length attribute**
CHAR(*n*) or VARCHAR(*n*)	MIN (VARBINARY(12*n*), 32704)
GRAPHIC(*n*) or VARGRAPHIC(*n*)	MIN (VARBINARY(12*n*), 32704)

Regardless of whether the length is specified, the length of the collation key must be less than or equal to the length attribute of the result. The actual result length of the collation key is approximately six times of the length of *string-expression* where the length of *string-expression* is in Unicode byte

representation. For certain *collation-name* such as UCA410_LKO_RKR (for Korean collation) the default length attribute of the result, 12*n*, might not be large enough and an error will be returned. To avoid such an error, the length attribute of the result must be explicitly specified to a larger constant. For the proper length attribute of the result, see *z/OS Support for Unicode: Using Conversion Services* for information about target buffer length considerations for Collation Services.

The result can be null; if the first argument is null, the result is the null value.

The COLLATION_KEY function uses Unicode Collation Services in z/OS to return the collation key. Unicode Collation Services support two collation versions:

- UCA400R1. This Collation version support Unicode standard character suite 4.0.0 and use Normalization Service under 4.0.1 Unicode character suite.
- UCA410. This Collation version support Unicode standard character suite 4.1.0 and use Normalization Service under 4.1.0 Unicode character suite.

If Unicode Collation Services are not available when the COLLATION_KEY function is run, an error is returned.

Example 1: The following query orders the employees by their surnames using the default Unicode Collation Algorithm V4.0.1(UCA), ignoring spaces, punctuation, and symbols, using Swedish linguistic conventions, and not comparing case:

```
SELECT FIRSTNAME, LASTNAME
  FROM DSN8B10.EMP
  ORDER BY COLLATION_KEY(LASTNAME, 'UCA400R1_AS_LSV_S2');
```

Example 2: The following query uses the COLLATION_KEY function on the LASTNAME column and the SALES_PERSON column to obtain the sort keys from the same collation name in order to do a culturally correct comparison. It finds the departments of employees in Quebec:

```
SELECT E.WORKDEPT
  FROM EMPLOYEE AS E INNER JOIN SALES AS S
  ON COLLATION_KEY(E.LASTNAME, 'UCA400R1_LFR') =
                  COLLATION_KEY(S.SALES_PERSON, 'UCA400R1_LFR')
  WHERE S.REGION = 'Quebec';
```

Example 3: Create an index EMPLOYEE_NAME_SORT_KEY for table EMPLOYEE based on built-in function COLLATION_KEY with collation name 'UCA410_LDE' tailored for German.

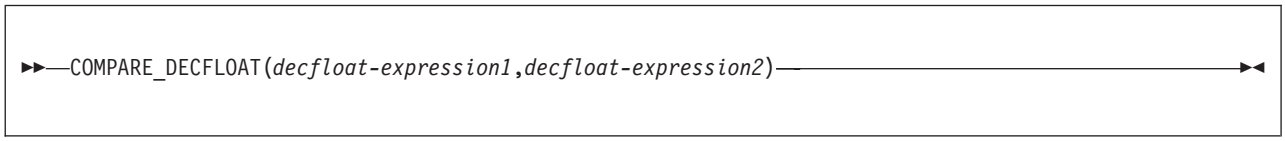
```
CREATE INDEX EMPLOYEE_NAME_SORT_KEY
  ON EMPLOYEE (COLLATION_KEY(LASTNAME, 'UCA410_LDE', 600),
               COLLATION_KEY(FIRSTNAME, 'UCA410_LDE', 600),
               ID);
```

Related reference:

Description of parameters in area CUNBOPR (z/OS: Unicode Services User's Guide and Reference)

COMPARE_DECFLOAT

The COMPARE_DECFLOAT function returns a SMALLINT value that indicates whether the two arguments are equal or unordered, or whether one argument is greater than the other.



The schema is SYSIBM.

decfloat-expression1
An expression that returns a DECFLOAT value.

decfloat-expression2
An expression that returns a DECFLOAT value.

decfloat-expression1 is compared with *decfloat-expression2* and the result is returned according to the following rules:

- If both arguments are finite, the comparison is algebraic and follows the procedure for DECFLOAT subtraction. If the different is exactly zero with either sign, the arguments are equal. If a nonzero difference is positive, the first argument is greater than the second argument. If a nonzero difference is negative, the first argument is less than the second.
- Positive zero and negative zero compare as equal.
- Positive infinity compares equal to positive infinity.
- Positive infinity compares greater than any finite number.
- Negative infinity compares equal to negative infinity.
- Negative infinity compares less than any finite number.
- Numeric comparison is exact and the result is determined for finite operands as if range and precision were unlimited. Overflow or underflow cannot occur.
- If either argument is NaN or sNaN (positive or negative), the result is unordered.

Numeric comparison is exact, and the result is determined for finite operands as if the range and precision were unlimited. An overflow or underflow condition cannot occur.

If one argument is DECFLOAT(16) and the other is DECFLOAT(34), the DECFLOAT(16) value is converted to DECFLOAT(34) before the comparison is made.

The arguments can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34).

One of the following values will be the result:
0 The arguments are exactly equal
1 *decfloat-expression1* is less than *decfloat-expression2*
2 *decfloat-expression1* is greater than *decfloat-expression2*
3 The arguments are unordered

The result of the function is a SMALLINT value.

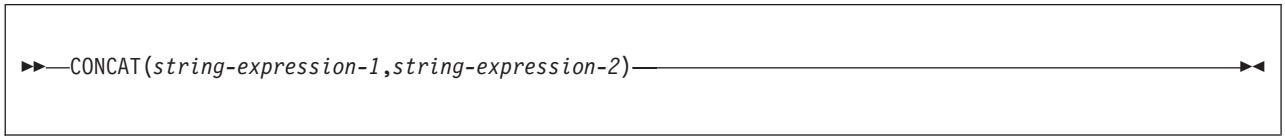
The result can be null; if any argument is null, the result is the null value.

Examples: The following examples demonstrate the values that will be returned when the function is used:

```
COMPARE_DECFLOAT(DECFLOAT(2.17), DECFLOAT(2.17))        = 0
COMPARE_DECFLOAT(DECFLOAT(2.17), DECFLOAT(2.170))       = 2
COMPARE_DECFLOAT(DECFLOAT(2.170), DECFLOAT(2.17))       = 1
COMPARE_DECFLOAT(DECFLOAT(2.17), DECFLOAT(0.0))         = 2
COMPARE_DECFLOAT(INFINITY,INFINITY)                     = 0
COMPARE_DECFLOAT(INFINITY,-INFINITY)                    = 2
COMPARE_DECFLOAT(DECFLOAT(-2),INFINITY)                 = 1
COMPARE_DECFLOAT(NAN,NAN)                               = 3
COMPARE_DECFLOAT(DECFLOAT(-0.1),SNAN)                   = 3
```

CONCAT

The CONCAT function combines two compatible string arguments.



The schema is SYSIBM.

The arguments must be compatible strings. For more information on compatibility, refer to the compatibility matrix in Table 23 on page 121.

Either argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.

The result of the function is a string that consists of the first string followed by the second string.

The result can be null; if any argument is null, the result is the null value.

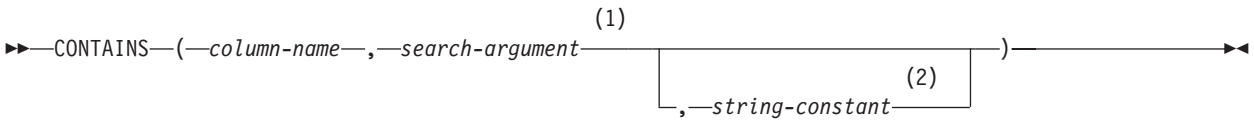
The CONCAT function is identical to the CONCAT operator. For more information, see "Expressions with the concatenation operator" on page 250.

Example: Using sample table DSN8B10.EMP, concatenate column FIRSTNME with column LASTNAME. Both columns are defined as varying-length character strings.

```
SELECT CONCAT(FIRSTNME, LASTNAME)
  FROM DSN8B10.EMP;
```

CONTAINS

The CONTAINS function searches a text search index using criteria that are specified in a search argument and returns a result about whether or not a match was found.



Notes:

1 The SQL statement that invokes the CONTAINS function can be dynamically prepared by using a typed parameter marker for the *search-argument*, as in the following example: `CONTAINS(C1,CAST(? AS CHAR(10)))`.

2 *string-constant* must conform to the rules for the *search-argument-options*.

search-argument-options:

Notes:

1 The same clause must not be specified more than once.

The schema is SYSIBM.

column-name
Specifies a qualified or unqualified name of a column that has a text search index that is to be searched. The column must exist in the table or view that is identified in the FROM clause in the statement and the column of the table, or the column of the underlying base table of the view must have an associated text search index. The underlying expression of the column of a view must be a simple column reference to the column of an underlying table, directly or through another nested view.

search-argument
Specifies an expression that returns a value that is a string value (except a LOB) that contains the terms to be searched for and must not be all blanks or the empty string. The actual length of the string must not exceed 4096 Unicode characters. The value is converted to Unicode before it is used to search the text search index. The maximum number of terms per query must not exceed 1024.

string-constant
Identifies a string constant that specifies the search argument options that are in effect for the function.

The options that can be specified as part of the *search-argument-options* are as follows:

QUERYLANGUAGE = *value*
Specifies the query language. The value can be any of the supported language codes. If the QUERYLANGUAGE option is not specified, the default is the language value of the text search index that is used when this function is invoked. If the language value of the text search index is AUTO, the default value for QUERYLANGUAGE is en_US.

RESULTLIMIT = *value*
Specifies the maximum number of results to be returned from the underlying search engine. The *value* can be an integer value between 1 and 2 147 483 647. If the RESULTLIMIT option is not specified, no result limit is in effect for the query.

This scalar function cannot be called for each row of the result table, depending on the plan that the optimizer chooses. This function can be called once for the query to the underlying search engine, and a result set of all of the primary keys that match are returned from the search engine. This result set is then joined to the table containing the column to identify the result rows. In this case, the RESULTLIMIT value acts like a FETCH FIRST ?? ROWS from the underlying text search engine and can be used as an optimization. If the search engine is called for each row of the result because the optimizer determines that is the best plan, then the RESULTLIMIT option has no effect. Also, the RESULTLIMIT option has no effect when the CONTAINS function is used along with the comparison operators (<, >, <=, and >=) or the equality operator (=) and a value of 0 (zero).

SYNONYM = *OFF* **or SYNONYM =** *ON*
Specifies whether to use a synonym dictionary that is associated with the text search index. Use the Synonym Tool to add a synonym dictionary to the collection. The default is OFF.

OFF Do not use a synonym dictionary.

ON Use the synonym dictionary that is associated with the text search index.

The result of the function is a large integer. If the second argument can be null, the result can be null. If the second argument is null, the result is the null value. If the third argument is null, the result is as if the third argument was not specified.

The result is 1 if the document contains a match for the search criteria that are specified in the search argument. Otherwise, the result is 0.

CONTAINS is a non-deterministic function.

Examples

Example 1: The following statement finds all of the employees who have "COBOL" in their resume. The text search argument is not case-sensitive.

```
SELECT EMPNO
 FROM EMP_RESUME
 WHERE RESUME_FORMAT = 'ascii'
  AND CONTAINS(RESUME, 'cobol') = 1
```

Example 2: The search argument does not need to be a string constant. The search argument can be any SQL string expression, including a string contained in a host variable.

The following statement searches for the exact term "ate" in the COMMENT column:

```
char search_arg[100]; /* input host variable */
...
EXEC SQL DECLARE C3 CURSOR FOR
   SELECT CUSTKEY
   FROM K55ADMIN.CUSTOMERS
   WHERE CONTAINS(COMMENT, :search_arg)= 1
   ORDER BY CUSTKEY;
strcpy(search_arg, "ate");
EXEC SQL OPEN C3;
...
```

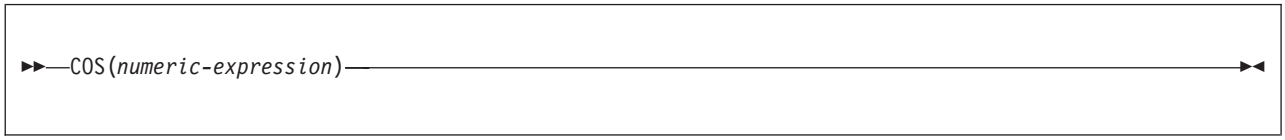
Example 3: The following statement finds 10 students at random who wrote online essays that contain the phrase "fossil fuel" in Spanish, which is "combustible fósil." These students will be invited for a radio interview.

Use the synonym dictionary that was created for the associated text search index. Because only 10 students are needed, you can optimize the query by using the RESULTLIMIT option to limit the number of results from the underlying text search server.

```
SELECT FIRSTNME, LASTNAME
   FROM STUDENT_ESSAYS
   WHERE CONTAINS(TERM_PAPER, 'combustible fósil',
      'QUERYLANGUAGE= es_ES RESULTLIMIT  = 10 SYNONYM=ON') = 1
```

COS

The COS function returns the cosine of the argument, where the argument is an angle, expressed in radians. The COS and ACOS functions are inverse operations.



The schema is SYSIBM.

The argument must be an expression that returns the value of any built-in numeric data type that is not DECFLOAT. If the argument is not a double precision floating-point number, it is converted to one for processing by the function.

The result of the function is a double precision floating-point number.

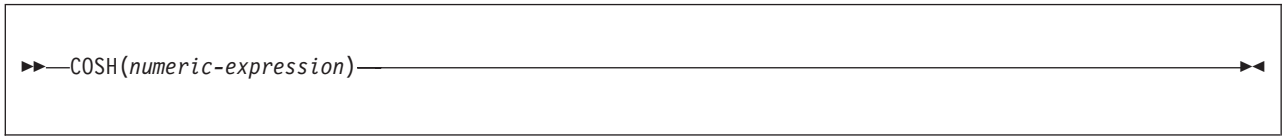
The result can be null; if the argument is null, the result is the null value.

Example: Assume that host variable COSINE is DECIMAL(2,1) with a value of 1.5. The following statement returns a double precision floating-point number with an approximate value of 0.07:

```
SELECT COS(:COSINE)
  FROM SYSIBM.SYSDUMMY1;
```

COSH

The COSH function returns the hyperbolic cosine of the argument, where the argument is an angle, expressed in radians.



The schema is SYSIBM.

The argument must be an expression that returns the value of any built-in numeric data type that is not DECFLOAT. If the argument is not a double precision floating-point number, it is converted to one for processing by the function.

The result of the function is a double precision floating-point number.

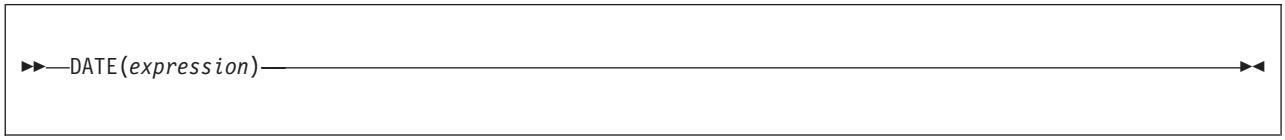
The result can be null; if the argument is null, the result is the null value.

Example: Assume that host variable HCOS is DECIMAL(2,1) with a value of 1.5. The following statement returns a double precision floating-point number with an approximate value of 2.35:

```
SELECT COSH(:HCOS)
  FROM SYSIBM.SYSDUMMY1;
```

DATE

The DATE function returns a date that is derived from a value.



The schema is SYSIBM.

The argument must be an expression that returns one of the following built-in data types: a date, a timestamp, a character string, a graphic string, or any numeric data type.

- If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and it must have one of the following values:
  - A valid string representation of a date or timestamp with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 101.
  - A character or graphic string with an actual length of 7 that represents a valid date in the form *yyyynnn*, where *yyyy* are digits denoting a year and *nnn* are digits between 001 and 366 denoting a day of that year.
- If *expression* is a number, it must be greater than or equal to one and less than or equal to 3652059.

If expression is not a DATE value, *expression* is cast as follows:

- If *expression* is a TIMESTAMP WITH TIME ZONE value, *expression* is cast to TIMESTAMP WITHOUT TIME ZONE, with the same precision as *expression*.
- If *expression* is a string, *expression* is cast to DATE.

The result of the function is a date.

The result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

If the argument is a timestamp, the result is the date part of the timestamp.

If the argument is a date, the result is that date.

If the argument is a number, the result is the date that is *n*-1 days after January 1, 0001, where *n* is the integral part of the number.

If the argument is a string, the result is the date that is represented by the string. If the string contains a time zone, the time zone is ignored. If the CCSID of the string is not the same as the corresponding default CCSID at the server, the string is first converted to that CCSID.

The result CCSID is the appropriate CCSID of the argument encoding scheme and the result subtype is the appropriate subtype of the CCSID.

Example 1: Assume that RECEIVED is a TIMESTAMP column in some table, and that one of its values is equivalent to the timestamp '1988-12-25-17.12.30.000000'. For this value, the following statement returns the internal representation of 25 December 1988.

```
DATE(RECEIVED)
```

Example 2: Assume that DATCOL is a CHAR(7) column in some table, and that one of its values is the character string '1989061'. For this value, the following statement returns the internal representation of 2 March 1989.

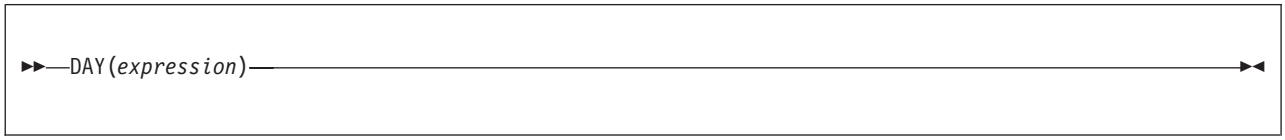
```
DATE(DATCOL)
```

Example 3: DB2 recognizes '1989-03-02' as the ISO representation of 2 March 1989. So, the following statement returns the internal representation of 2 March 1989.

```
DATE('1989-03-02')
```

DAY

The DAY function returns the day part of a value.



The schema is SYSIBM.

The argument must be an expression that returns one of the following built-in data types: a date, a timestamp, a character string, a graphic string, or any numeric data type.

- If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 101.
- If *expression* is a number, it must be a date duration or a timestamp duration. For the valid formats of datetime durations, see "Datetime operands" on page 147.

If *expression* is a timestamp with a time zone value, or a valid string representation of a timestamp with a time zone, the result is determined from the UTC representation of the datetime value.

The result of the function is a large integer.

The result can be null; if the argument is null, the result is the null value.

The other rules for the function depend on the data type of the argument:

If the argument is a date, timestamp, or string representation of either, the result is the day part of the value, which is an integer between 1 and 31.

If the argument is a date duration or timestamp duration, the result is the day part of the value, which is an integer between -99 and 99. A nonzero result has the same sign as the argument.

If the argument contains a time zone, the result is the year part of the value expressed in UTC.

Example 1: Set the INTEGER host variable DAYVAR to the day of the month on which employee 140 in the sample table DSN8B10.EMP was hired.

```
EXEC SQL SELECT DAY(HIREDATE)
  INTO :DAYVAR
  FROM DSN8B10.EMP
  WHERE EMPNO = '000140';
```

Example 2: Assume that DATE1 and DATE2 are DATE columns in the same table. Assume also that for a given row in this table, DATE1 and DATE2 represent the dates 15 January 2000 and 31 December 1999, respectively. Then, for the given row:

```
DAY(DATE1 - DATE2)
```

returns the value 15.

Example 3: The following invocations of the DAY function all return the same result:

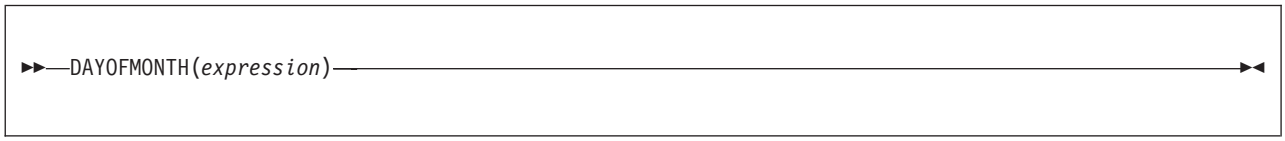
```
SELECT DAY('2003-01-02-20.00.00'),
  DAY('2003-01-02-12.00.00-08:00'),
  DAY('2003-01-03-05.00.00+09:00')
FROM SYSIBM.SYSDUMMY1;
```

For each invocation of the DAY function in this SELECT statement, the result is 2.

When the input argument contains a time zone, the result is determined from the UTC representation of the input value. The string representations of a timestamp with a time zone in the SELECT statement all have the same UTC representation: 2003-01-02-20.00.00. The day portion of the UTC representation is 2.

DAYOFMONTH

The DAYOFMONTH function returns the day part of a value. The function is similar to the DAY function, except DAYOFMONTH does not support a date or timestamp duration as an argument.



The schema is SYSIBM.

The argument must be an expression that returns a value of a date, a timestamp, a character string, or a graphic string built-in data type.

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 101.

If *expression* is a timestamp with a time zone value, or a valid string representation of a timestamp with a time zone, the result is determined from the UTC representation of the datetime value.

The result of the function is a large integer between 1 and 31, which represents the day part of the value.

The result can be null; if the argument is null, the result is the null value.

Example 1: Set the INTEGER variable DAYVAR to the day of the month on which employee 140 in sample table DSN8B10.EMP was hired.

```
SELECT DAYOFMONTH(HIREDATE)
  INTO :DAYVAR
  FROM DSN8B10.EMP
  WHERE EMPNO = '000140';
```

Example 2: The following invocations of the DAYOFMONTH function returns the same result:

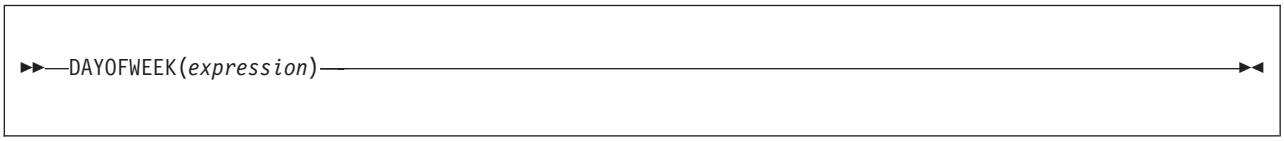
```
SELECT DAYOFMONTH('2003-01-02-20.00.00'),
   DAYOFMONTH('2003-01-02-12.00.00-08:00'),
   DAYOFMONTH('2003-01-03-05.00.00+09:00')
  FROM SYSIBM.SYSDUMMY1;
```

For each invocation of the DAYOFMONTH function in this SELECT statement, the result is 2.

When the input argument contains a time zone, the result is determined from the UTC representation of the input value. The string representations of a timestamp with a time zone in the SELECT statement all have the same UTC representation: 2003-01-02-20.00.00. The day portion of the UTC representation is 2.

DAYOFWEEK

The DAYOFWEEK function returns an integer, in the range of 1 to 7, that represents the day of the week, where 1 is Sunday and 7 is Saturday. The DAYOFWEEK function is similar to the DAYOFWEEK_ISO function.



The schema is SYSIBM.

The argument must be an expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, or a graphic string.

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 101.

If *expression* is a timestamp with a time zone, or a valid string representation of a timestamp with a time zone, the result is determined from the UTC representation of the datetime value.

The result of the function is a large integer.

The result can be null; if the argument is null, the result is the null value.

Example 1: Using sample table DSN8B10.EMP, set the integer host variable DAY_OF_WEEK to the day of the week that Christine Haas (EMPNO = '000010') was hired (HIREDATE).

```
SELECT DAYOFWEEK(HIREDATE)
  INTO :DAY_OF_WEEK
  FROM DSN8B10.EMP
  WHERE EMPNO = '000010';
```

The result is that DAY_OF_WEEK is set to 6, which represents Friday.

Example 2: The following query returns four values: 1, 2, 1, and 2.

```
SELECT DAYOFWEEK(CAST('10/11/1998' AS DATE)),
       DAYOFWEEK(TIMESTAMP('10/12/1998', '01.02')),
       DAYOFWEEK(CAST(CAST('10/11/1998' AS DATE) AS CHAR(20))),
       DAYOFWEEK(CAST(TIMESTAMP('10/12/1998', '01.02') AS CHAR(26)))
  FROM SYSIBM.SYSDUMMY1;
```

Example 3: The following invocations of the DAYOFWEEK function returns the same result:

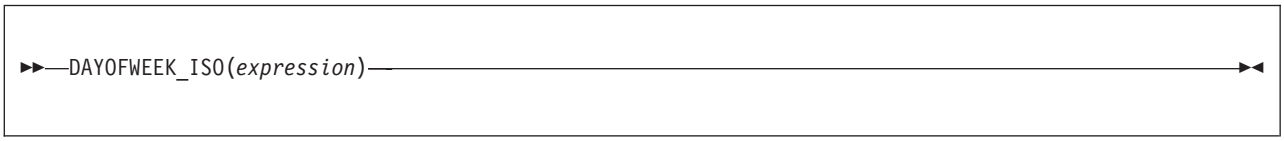
```
SELECT DAYOFWEEK('2003-01-02-20.00.00'),
   DAYOFWEEK('2003-01-02-12.00.00-08:00'),
   DAYOFWEEK('2003-01-03-05.00.00+09:00')
  FROM SYSIBM.SYSDUMMY1;
```

For each invocation of the DAYOFWEEK function in this SELECT statement, the result is 5 (Sunday is considered the first day of the week).

When the input argument contains a time zone, the result is determined from the UTC representation of the input value. The string representations of a timestamp with a time zone in the SELECT statement all have the same UTC representation: 2003-01-02-20.00.00.

DAYOFWEEK_ISO

The DAYOFWEEK_ISO function returns an integer, in the range of 1 to 7, that represents the day of the week, where 1 is Monday and 7 is Sunday. The DAYOFWEEK_ISO function is similar to the DAYOFWEEK function.



The schema is SYSIBM.

The argument must be an expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, or graphic string.

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 101.

If *expression* is a timestamp with a time zone, or a valid string representation of a timestamp with a time zone, the result is determined from the UTC representation of the datetime value.

The result of the function is a large integer.

The result can be null; if the argument is null, the result is the null value.

Example 1: Using sample table DSN8B10.EMP, set the integer host variable DAY_OF_WEEK to the day of the week that Christine Haas (EMPNO = '000010') was hired (HIREDATE).

```
SELECT DAYOFWEEK_ISO(HIREDATE)
  INTO :DAY_OF_WEEK
  FROM DSN8B10.EMP
  WHERE EMPNO = '000010';
```

The result is that DAY_OF_WEEK is set to 5, which represents Friday.

Example 2: The following query returns four values: 7, 1, 7, and 1.

```
SELECT DAYOFWEEK_ISO(CAST('10/11/1998' AS DATE)),
       DAYOFWEEK_ISO(TIMESTAMP('10/12/1998', '01.02')),
       DAYOFWEEK_ISO(CAST(CAST('10/11/1998' AS DATE) AS CHAR(20))),
       DAYOFWEEK_ISO(CAST(TIMESTAMP('10/12/1998', '01.02') AS CHAR(26)))
  FROM SYSIBM.SYSDUMMY1;
```

Example 3: The following list shows what is returned by the DAYOFWEEK_ISO function for various dates.

```
DATE:          DAYOFWEEK_ISO returns:
2003-12-28     '7'
2003-12-31     '3'
2004-01-01     '4'
2004-01-10     '6'
```

```
   2005-01-04      '2'
   2005-12-31      '7'
   2006-01-01      '7'
   2006-01-03      '2'
```

Example 4: The following invocations of the DAYOFWEEK_ISO function returns the same result:

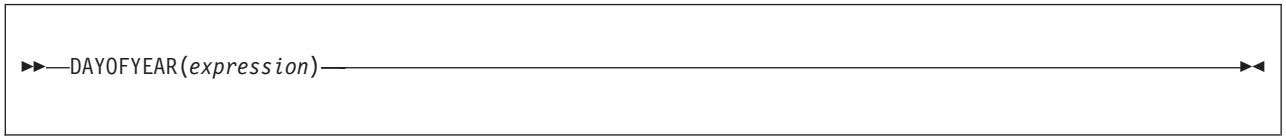
```
SELECT DAYOFWEEK_ISO('2003-01-02-20.00.00'),
   DAYOFWEEK_ISO('2003-01-02-12.00.00-08:00'),
   DAYOFWEEK_ISO('2003-01-03-05.00.00+09:00')
  FROM SYSIBM.SYSDUMMY1;
```

For each invocation of the DAYOFWEEK_ISO function in this SELECT statement, the result is 4 (Monday is considered the first day of the week).

When the input argument contains a time zone, the result is determined from the UTC representation of the input value. The string representations of a timestamp with a time zone in the SELECT statement all have the same UTC representation: 2003-01-02-20.00.00.

DAYOFYEAR

The DAYOFYEAR function returns an integer, in the range of 1 to 366, that represents the day of the year, where 1 is January 1.



The schema is SYSIBM.

The argument must be an expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, or a graphic string.

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 101.

If *expression* is a timestamp with a time zone value, or a valid string representation of a timestamp with a time zone, the result is determined from the UTC representation of the datetime value.

The result of the function is a large integer.

The result can be null; if the argument is null, the result is the null value.

Example 1: Using sample table DSN8B10.EMP, set the integer host variable AVG_DAY_OF_YEAR to the average of the day of the year on which employees were hired (HIREDATE):

```
SELECT AVG(DAYOFYEAR(HIREDATE))
  INTO :AVG_DAY_OF_YEAR
  FROM DSN8B10.EMP;
```

The result is that AVG_DAY_OF_YEAR is set to 202.

Example 2: The following invocations of the DAYOFYEAR function returns the same result:

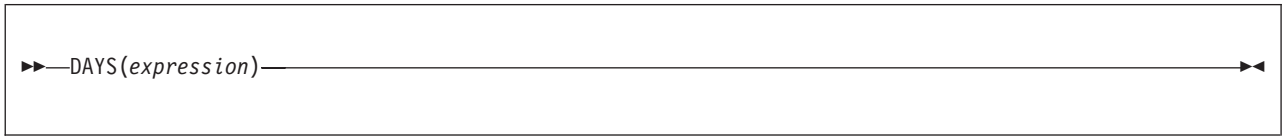
```
SELECT DAYOFYEAR('2003-01-02-20.00.00'),
   DAYOFYEAR('2003-01-02-12.00.00-08:00'),
   DAYOFYEAR('2003-01-03-05.00.00+09:00')
  FROM SYSIBM.SYSDUMMY1;
```

The results for this SELECT statement are 2 ,2 ,3.

When the input argument contains a time zone, the result is determined from the UTC representation of the input value. The string representations of a timestamp with a time zone in the SELECT statement all have the same UTC representation: 2003-01-02-20.00.00.

DAYS

The DAYS function returns an integer representation of a date.



The schema is SYSIBM.

The argument must be an expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, or a graphic string.

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 101.

If *expression* is a timestamp with a time zone value, or a valid string representation of a timestamp with a time zone, the result is determined from the UTC representation of the datetime value.

The result of the function is a large integer.

The result can be null; if the argument is null, the result is the null value.

The result is 1 more than the number of days from January 1, 0001 to *D*, where *D* is the date that would occur if the DATE function were applied to the argument.

Example1: Set the INTEGER host variable DAYSVAR to the number of days that employee 140 had been with the company on the last day of 1997.

```
EXEC SQL SELECT DAYS('1997-12-31') - DAYS(HIREDATE) + 1
  INTO :DAYSVAR
  FROM DSN8B10.EMP
  WHERE EMPNO = '000140';
```

Example 2: The following invocations of the DAYS function returns the same result:

```
SELECT DAYS('2003-01-02-20.00.00'),
   DAYS('2003-01-02-12.00.00-08:00'),
   DAYS('2003-01-03-05.00.00+09:00')
  FROM SYSIBM.SYSDUMMY1;
```

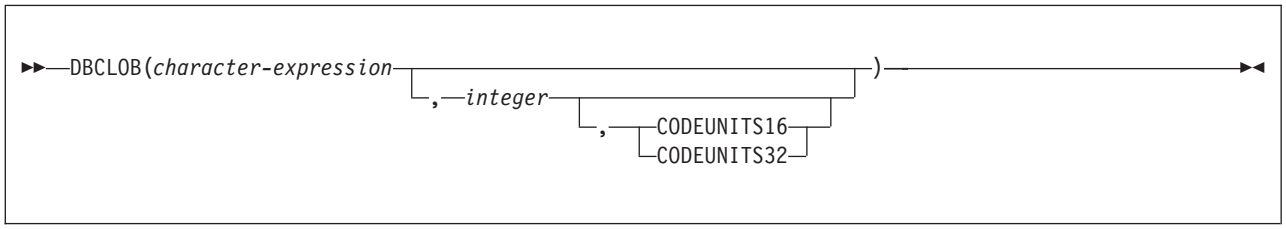
For each invocation of the DAYS function in this SELECT statement, the result is 731217.

When the input argument contains a time zone, the result is determined from the UTC representation of the input value. The string representations of a timestamp with a time zone in the SELECT statement all have the same UTC representation: 2003-01-02-20.00.00.

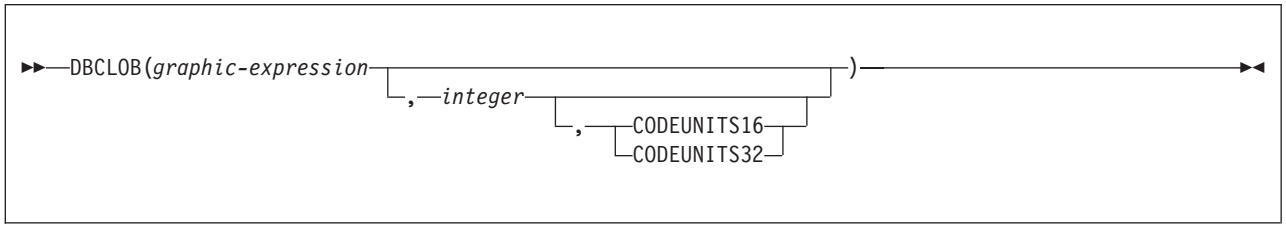
DBCLOB

The DBCLOB function returns a DBCLOB representation of a character string value (with the single-byte characters converted to double-byte characters) or a graphic string value.

Character to DBCLOB:



Graphic to DBCLOB:



The schema is SYSIBM.

Character to DBCLOB

character-expression
An expression that returns a value that is an EBCDIC-encoded or Unicode-encoded character string. It cannot be BIT data. The argument does not need to be mixed data, but any occurrences of X'0E' and X'0F' in the string must conform to the rules for EBCDIC mixed data. (See "Character strings" on page 84 for these rules.)

integer
The length attribute of the resulting DBCLOB. The value of *integer* must be between 1 and the maximum length of a DBCLOB, expressed in the units that are either implicitly or explicitly specified.

If CODEUNITS16 or CODEUNITS32 is specified, see "Determining the length attribute of the final result" on page 90 for information about how to calculate the length attribute of the result string. If CODEUNITS32 is specified, the value of *integer* must be between 1 and the maximum length of a DBCLOB divided by two (to allow for an intermediate result string that is long enough to evaluate the function).

If *integer* is not specified and *character-expression* is an empty string constant, the length attribute of the result is 1, and the result is an empty string. Otherwise, the length attribute of the result is the same as the length attribute of *character-expression*.

CODEUNITS16 or CODEUNITS32
Specifies the unit that is used to express *integer*. If CODEUNITS16 or CODEUNITS32 is specified, the input is EBCDIC, and there is no system CCSID for EBCDIC GRAPHIC data, an error occurs.

CODEUNITS16
Specifies that *integer* is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32
Specifies that *integer* is expressed in terms of 32-bit UTF-32 code units.

For more information about CODEUNITS16 and CODEUNITS32, see "String unit specifications" on page 87.

The actual length of the result is the minimum of the length attribute of the result and the actual length of *character-expression*. If the length of *character-expression*, as measured in single-byte characters, is greater than the specified length of the result, as measured in double-byte characters, the result is truncated. Unless all the truncated characters are blanks appropriate for *character-expression*, a warning is returned.

The CCSID of the result is the graphic CCSID that corresponds to the character CCSID of *character-expression*.

For EBCDIC input data, each character of *character-expression* determines a character of the result. The argument might need to be converted to the native form of mixed data before the result is derived. Let *M* denote the system CCSID for mixed data. The argument is not converted if any of the following conditions is true:

- The argument is mixed data and its CCSID is *M*.
- The argument is SBCS data and its CCSID is the same as the system CCSID for SBCS data. In this case, the operation proceeds as if the CCSID of the argument is *M*.

Otherwise, the argument is a new string *S* derived by converting the characters to the coded character set identified by *M*. If there is no system CCSID for mixed data, conversion is to the coded character set that the system CCSID for SBCS data identifies.

The result is derived from *S* using the following steps:
- Each shift character (X'0E' or X'0F') is removed.
- Each double-byte character remains as is.
- Each single-byte character is replaced by a double-byte character.

The replacement for a single-byte character is the equivalent DBCS character if an equivalent exists. Otherwise, the replacement is X'FEFE'. The existence of an equivalent character depends on *M*. If there is no system CCSID for mixed data, the DBCS equivalent of X'xx' for EBCDIC is X'42xx', except for X'40', whose DBCS equivalent is X'4040'.

For Unicode input data, each character of *character-expression* determines a character of the result. The argument might need to be converted to the native form of mixed data before the result is derived. Let *M* denote the system CCSID for mixed data. The argument is not converted if any of the following conditions is true:

- The argument is mixed data, and its CCSID is *M*.
- The argument is SBCS data, and its CCSID is the same as the system CCSID for SBCS data. In this case, the operation proceeds as if the CCSID of the argument is *M*.

Otherwise, the argument is a new string *S* derived by converting the characters to the coded character set identified by *M*.

The result is derived from *S* using the following steps:
- Each non-supplementary character is replaced by a Unicode double-byte character (a UTF-16 code point). A non-supplementary character in UTF-8 is between 1 and 3 bytes.
- Each supplementary character is replaced by a pair of Unicode double-byte characters (a pair of UTF-16 code points).

The replacement for a single-byte character is the Unicode equivalent character if an equivalent exists. Otherwise, the replacement is X'FFFD'.

Graphic to DBCLOB

graphic-expression
An expression that returns a value that is an EBCDIC-encoded or Unicode-encoded graphic string.

integer
The length attribute for the resulting varying-length graphic string. The value must be an integer between 1 and the maximum length of a DBCLOB, expressed in the units that are either implicitly or explicitly specified.

If CODEUNITS16 or CODEUNITS32 is specified, see "Determining the length attribute of the final result" on page 90 for information about how to calculate the length attribute of the result string.

If *integer* is not specified and *graphic-expression* is an empty string constant, the length attribute of the result is 1, and the result is an empty string. Otherwise, the length attribute of the result is the same as the length attribute of *graphic-expression*.

CODEUNITS16 or CODEUNITS32
Specifies the unit that is used to express *integer*. If CODEUNITS16 or CODEUNITS32 is specified, the input is EBCDIC, and there is no system CCSID for EBCDIC GRAPHIC data, an error occurs.

CODEUNITS16
Specifies that *integer* is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32
Specifies that *integer* is expressed in terms of 32-bit UTF-32 code units.

For more information about CODEUNITS16 and CODEUNITS32, see "String unit specifications" on page 87.

The actual length of the result is the minimum of the length attribute of the result and the actual length of *graphic-expression*. If the length of *graphic-expression* is greater than the length attribute of the result, truncation is performed. Unless all of the truncated characters are double-byte blanks, a warning is returned.

The CCSID of the result is the same as the CCSID of *graphic-expression*.

The result of the function is a DBCLOB.

The result can be null; if the first argument is null, the result is the null value.

The length attribute and actual length of the result are measured in double-byte characters because the result is a graphic string.

Example 1: Assume that the application encoding scheme is Unicode. The following statement returns a graphic (UTF-16) host variable.

```
VALUES DBCLOB('123')
  INTO :GHV1;
```

Example 2: FIRSTNAME is a VARCHAR(12) column (Unicode UTF-8 data) in table T1. One of its values is the 6-character string 'Jürgen'. When FIRSTNAME has this value:

```
Function ...                          Returns ...
DBCLOB(FIRSTNAME,3,CODEUNITS32)       'Jür'  -- x'004A00FC0072'
DBCLOB(FIRSTNAME,3,CODEUNITS16)       'Jür'  -- x'004A00FC0072'
```

DECFLOAT

The DECFLOAT function returns a decimal floating-point representation of either a number or a character string representation of a number, a decimal number, an integer, a floating-point number, or a decimal floating-point number.

Numeric to DECFLOAT:

String to DECFLOAT:

The schema is SYSIBM.

Numeric to DECFLOAT

numeric-expression
An expression that returns a value of any built-in numeric data type.

34 or 16
Specifies the number of digits of precision for the result. The default is 34.

String to DECFLOAT

string-expression
An expression that returns a value of a character or graphic string (except a CLOB or DBCLOB) with a length attribute that is not greater than 255 bytes. Leading and trailing blanks are eliminated, and the resulting string is folded to uppercase. The expression must conform to the rules for forming a floating-point, decimal floating-point, integer, or decimal constant.

Use the *string-expression* syntax variation to specify a negative zero as a constant, or to preserve the precision of a floating point constant.

34 or 16
Specifies the number of digits of precision for the result. The default is 34.

The result is the same number that would result from `CAST(`*`string-expression`*` AS DECFLOAT(n))` or `CAST(`*`numeric-expression`*` AS DECFLOAT(n))`. Leading and trailing blanks are removed from the string, and the resulting substring must conform to the rules for forming a string representation of an SQL decimal-floating point constant.

If necessary, the source is rounded to the precision of the target.

For static SQL statements other than CREATE VIEW, the ROUNDING bind option or the native SQL procedure option determines the rounding mode.

For dynamic SQL statements (and static CREATE VIEW statements), the special register CURRENT DECFLOAT ROUNDING MODE determines the rounding mode.

The result of the function is a DECFLOAT with the implicitly or explicitly specified number of digits of precision.

The result can be null; if the first argument is null, the result is the null value.

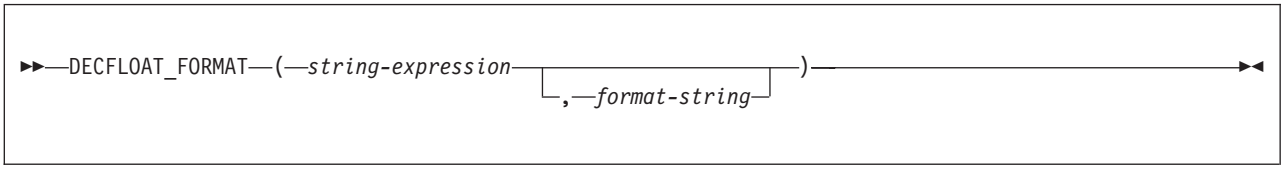
Note: To increase the portability of applications, use the CAST specification. For more information, see "CAST specification" on page 267.

Example: When a keyword is used for a special value that is expressed as a constant in a context where the keyword could be interpreted as a name, the DECFLOAT function can be used to explicitly cast the value to decimal-floating point. Assume that MYTAB contains columns C1 and SNAN, and that you want to reference the decimal floating-point value for infinity in the same SQL statement. Use the DECFLOAT function to explicitly cast SNAN as a decimal floating-point value to ensure that it is not interpreted as the name of a column, parameter or variable:

```
SELECT INFINITY                 -- column named SNAN
    FROM MYTAB
  WHERE C1 = DECFLOAT ('sNaN' ) -- comparison is made with the
                                -- decimal floating-point sNaN value
```

DECFLOAT_FORMAT

The DECFLOAT_FORMAT function returns a DECFLOAT(34) value that is based on the interpretation of the input string using the specified format.



The schema is SYSIBM.

string-expression
An expression that returns a value that is a CHAR and VARCHAR data type. If a supplied argument is a GRAPHIC or VARGRAPHIC data type, it is first converted to VARCHAR before evaluating the function. Leading and trailing blanks are removed from the string. If *format-string* is not specified, the resulting substring must conform to the rules for forming an SQL integer, decimal, floating-point, or decimal floating-point constant and not be greater than 42 bytes. Otherwise, the resulting substring must contain the components of a number that corresponds to the format specified by *format-string*.

format-string
An expression that returns a value that is a built-in character string data type. If a supplied argument is a graphic string (except DBCLOB), it is first converted to a character string before the function is evaluated. The actual length must not be greater than 254 bytes.

The value is a template for how *string-expression* is to be interpreted for conversion to a DECFLOAT value. *format-string* must contain a valid combination of the listed format elements according to the following rules:

- At least one '0' or '9' format element must be specified.
- A sign format element ('S', 'MI', 'PR') can be specified only one time.
- A decimal point format element can be specified only one time.
- Alphabetic format elements must be specified in upper case.
- A prefix format element can only be specified at the beginning of the format string, before any format elements that are not prefix format elements. When multiple prefix format elements are specified they can be specified in any order.
- A suffix format element can only be specified at the end of the format string, after any format elements that are not suffix format elements.
- A comma format element can be the first format element that is not a prefix format element. There can be any number of comma format elements.
- Blanks must not be specified between format elements. Leading and trailing blanks can be specified but are ignored.

Table 61. Format elements for the DECFLOAT_FORMAT function

Format element	Description
0	Represents a digit. A digit is expected if the '0' format element is to the left of the decimal point. Leading zeros must be specified if there are fewer digits to the left of the decimal point in the *string-expression* than in the *format-string*. A digit can be included if the '0' format element is to the right of the decimal point.
9	Represents a digit that can be included at the specified location.
S	**Prefix** If *string-expression* represents a negative number, a leading minus sign (–) is expected at the specified location. If *string-expression* represents a positive number, a leading plus sign (+) or leading blank can be included at the specified location.
$	**Prefix** A leading dollar sign ('$') is expected at the specified location.
MI	**Suffix** If *string-expression* represents a negative number, a trailing minus sign (–) is expected at the specified location. If *string-expression* represents a positive number, a trailing blank can be included at the specified location.
PR	**Suffix** If *string-expression* represents a negative number, a leading less than character (<) and a trailing greater than character (>) are expected. If *string-expression* represents a positive number, a leading blank and a trailing blank can be included.
,	Represents a group separator. A group separator is expected at the specified location if there is a character to the left of it that is not a prefix character.
.	A period represents a decimal point that is expected at the specified location.

If *format-string* is not specified, *string-expression* must conform to the rules for forming an SQL integer, decimal, floating-point, or decimal floating-point constant and have a length not greater than 42 bytes.

The result is a DECFLOAT(34).

The result can be null; if any argument is null, the result is the null value.

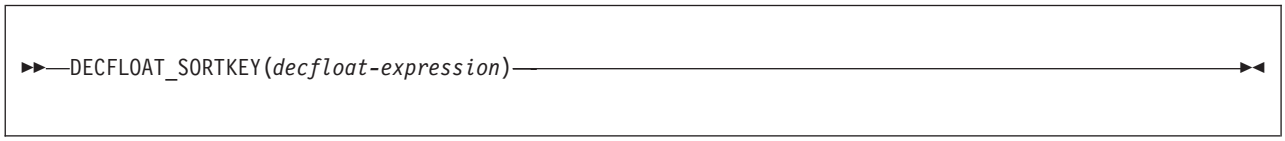
Syntax alternatives: TO_NUMBER is a synonym for DECFLOAT_FORMAT.

Table 62. Examples of DECFLOAT_FORMAT

Example	Result
`DECFLOAT_FORMAT( '123.45' )`	123.45
`DECFLOAT_FORMAT( '-123456.78' )`	-123456.78
`DECFLOAT_FORMAT( '+123456.78' )`	123456.78
`DECFLOAT_FORMAT( '1.23E4' )`	12300
`DECFLOAT_FORMAT( '123.4', '9999.99' )`	123.40
`DECFLOAT_FORMAT( '001,234', '000,000' )`	1234
`DECFLOAT_FORMAT( '1234 ', '9999MI' )`	1234
`DECFLOAT_FORMAT( '1234-', '9999MI' )`	-1234
`DECFLOAT_FORMAT( '+1234', 'S9999' )`	1234
`DECFLOAT_FORMAT( '-1234', 'S9999' )`	-1234
`DECFLOAT_FORMAT( ' 1234 ', '9999PR' )`	1234
`DECFLOAT_FORMAT( '<1234>', '9999PR' )`	-1234
`DECFLOAT_FORMAT( '$123,456.78', '$999,999.99' )`	123456.78

DECFLOAT_SORTKEY

The DECFLOAT_SORTKEY function returns a binary value that can be used when sorting DECFLOAT values. The sorting occurs in a manner that is consistent with the IEEE 754R specification on total ordering.



The schema is SYSIBM.

decfloat-expression
An expression that returns a DECFLOAT value.

decfloat-expression can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34).

The result is a fixed length binary string with a length attribute of 9 if *decfloat-expression* is a DECFLOAT(16) value or 17 if *decfloat-expression* is a DECFLOAT(34) value.

The result can be null; if the argument is null, the result is the null value.

Example: Assume that the following CREATE TABLE statement is used to create a table with a column that contains DECFLOAT values and the INSERT statements are used to populate the table:

```
CREATE TABLE T1(D1 DECFLOAT(16));
INSERT INTO T1 VALUES (2.100);
INSERT INTO T1 VALUES (2.10);
INSERT INTO T1 VALUES (2.1000);
INSERT INTO T1 VALUES (2.1);
```

Then the following SELECT statement is used to return the values from D1:

```
SELECT D1 FROM T1 ORDER BY D1;
```

The SELECT statement returns the following values, but because all numbers in the column have the same value, the ORDER BY clause has no effect and the values are returned in an arbitrary order:

```
D1
---------

2.1
2.1000
2.10
2.100
```

The following SELECT statement, which includes the DECFLOAT_SORTKEY function in the ORDER BY clause, returns the properly ordered values:

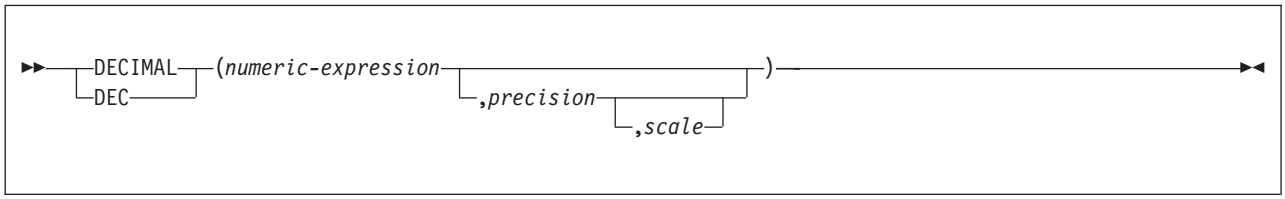
```
SELECT D1
   FROM T1
   ORDER BY (DECFLOAT_SORTKEY(D1));

D1
--------
```

```
2.1000
2.100
2.10
2.1
```

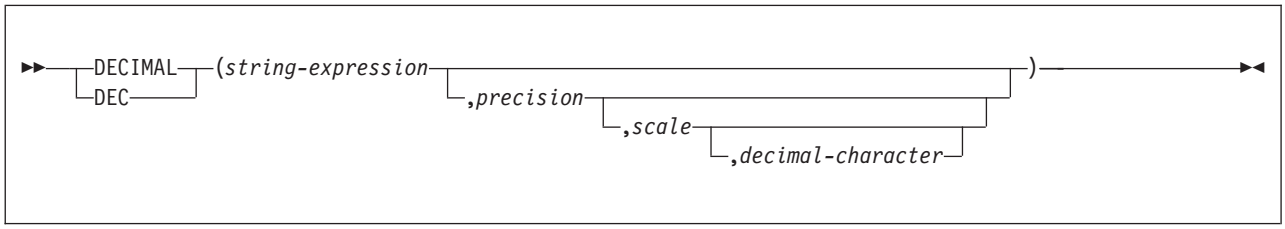
DECIMAL or DEC

The DECIMAL function returns a decimal representation of either a number or a character-string or graphic-string representation of a number, an integer, or a decimal number.

Numeric to Decimal:



String to Decimal:



The schema is SYSIBM.

Numeric to decimal

numeric-expression
An expression that returns a value of any built-in numeric data type.

precision
An integer constant with a value in the range of 1 to 31. The value of this second argument specifies the precision of the result.

The default value depends on the data type of the first argument as follows:
- 5 if the first argument is a small integer
- 11 if the first argument is a large integer
- 19 if the first argument is a big integer
- 31 if the first argument is a DECFLOAT value
- 15 in all other cases

scale
An integer constant that is greater than or equal to zero and less than or equal to *precision*. The value specifies the scale of the result. The default value is 0.

The result of the function is the same number that would occur if the argument were assigned to a decimal column or variable with precision *p* and scale *s*, where *p* and *s* are specified by the second and third arguments. An error occurs if the number of significant digits required to represent the whole part of the number is greater than *p-s*.

String to decimal

string-expression
An expression that returns a value of a character or graphic string (except a CLOB or DBCLOB) with a length attribute that is not greater than 255 bytes.

The string must contain a valid string representation of a number. Leading and trailing blanks are removed from the string, and the resulting substring must conform to the rules for forming a valid string representation of an SQL integer or decimal constant.

precision
An integer constant with a value in the range of 1 to 31. The value of this second argument specifies the precision of the result.

The default value depends on the data type of the first argument as follows:
- 5 if the first argument is a small integer
- 11 if the first argument is a large integer
- 15 in all other cases

scale
An integer constant that is greater than or equal to zero and less than or equal to *precision*. The value specifies the scale of the result. The default value is 0.

decimal-character
A single-byte character constant used to delimit the decimal digits in *string-expression* from the whole part of the number. The character cannot be a digit, plus (+), minus (-), or blank. The default value is period (.) or comma (,); the default value cannot be used in *string-expression* if a different value for *decimal-character* is specified.

The result is the same number that would result from `CAST(`*`string-expression`*` AS DECIMAL(`*`p`*`,`*`s`*`))`. Digits are truncated from the end of the decimal number if the number of digits to the right of the decimal separator character is greater than the scale *s*. An error is returned if the number of significant digits to the left of the decimal character (the whole part of the number) in *string-expression* is greater than *p-s*.

The result of the function is a decimal number with precision of *p* and scale of *s*, where *p* and *s* are the second and third arguments. If the first argument can be null, the result can be null; if the first argument is null, the result is null.

Note: To increase the portability of applications when the precision is specified, use the CAST specification. For more information, see "CAST specification" on page 267.

Example 1: Represent the average salary of the employees in DSN8B10.EMP as an 8-digit decimal number with two of these digits to the right of the decimal point.

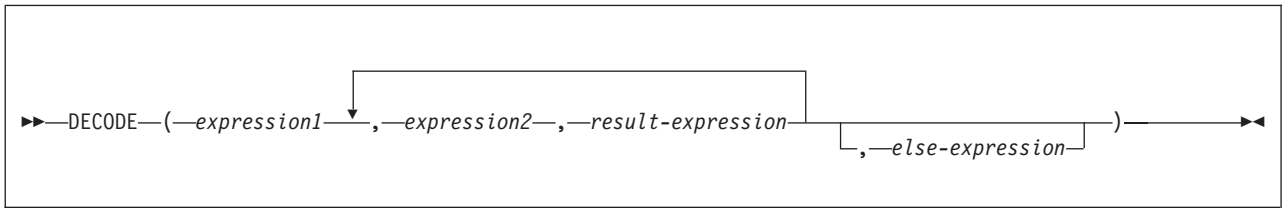
```
SELECT DECIMAL(AVG(SALARY),8,2)
  FROM DSN8B10.EMP;
```

Example 2: Assume that updates to the SALARY column are input as a character string that uses comma as the decimal character. For example, the user inputs 21400,50. The input value is assigned to the host variable NEWSALARY that is defined as CHAR(10), and the host variable is used in the following UPDATE statement:

```
UPDATE DSN8B10.EMP
  SET SALARY = DECIMAL (:NEWSALARY,9,2,',')
  WHERE EMPNO = :EMPID;
```

DECODE

The DECODE function compares each *expression2* to *expression1*. If *expression1* is equal to *expression2*, or both *expression1* and *expression2* are null, the value of the *result-expression* is returned. If no *expression2* matches *expression1*, the value of *else-expression* is returned. Otherwise a null value is returned.



The schema is SYSIBM.

The DECODE function is similar to the CASE expression, with the exception of how DECODE handles null values:

- A null value in *expression1* will match a corresponding null value in *expression2*.
- If the NULL keyword is used as an argument in the DECODE function, it must be case to a data type that is appropriate for comparison.

An argument of DECODE must not represent an array value.

The rules for determining the result type of the result of the DECODE function are based on the corresponding CASE expression.

The following table shows equivalent DECODE functions and CASE expressions. Both the DECODE function and the corresponding CASE expression achieve the same result.

Table 63. Equivalent DECODE functions and CASE expressions (each returns the same results)

DECODE function	CASE expression	Notes

`DECODE(c1, 7, 'a',`
`  6, 'b', 'c')`
	`CASE c1`
	`  WHEN 7 THEN 'a'`
	`  WHEN 6 THEN 'b'`
	`  ELSE 'c'`
	`END`	

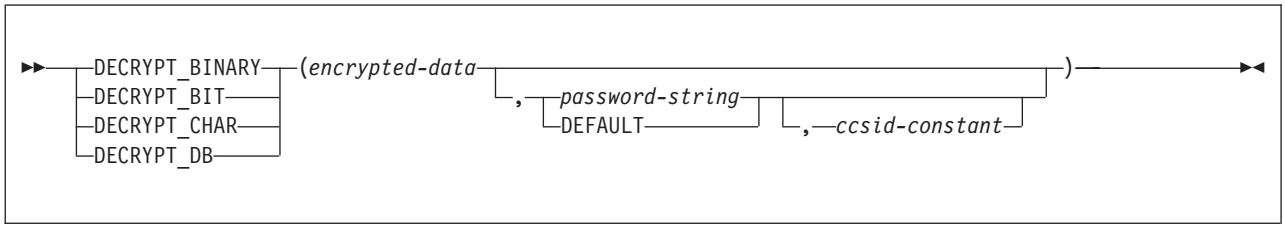
`DECODE(c1, var1, 'a',`
`  var2, 'b')`
	`CASE`
	`WHEN c1 = var1 OR`
	`  (c1 IS NULL AND`
	`  var1 ISNULL) THEN 'a'`
	`WHEN c1 = var2 OR`
	`  (c1 IS NULL AND`
	`  var2 ISNULL) THEN 'b'`
	`ELSE NULL`
	`END`
		The values of c1, var1, and var2 can be null values.

Table 63. Equivalent DECODE functions and CASE expressions (each returns the same results) (continued)

DECODE function	CASE expression	Notes
SELECT ID, DECODE(STATUS, 'A', 'Accepted', D', 'Denied', CAST(NULL AS VARCHAR(1)), 'Unknown', 'Other') FROM CONTRACTS	SELECT ID, CASE WHEN STATUS = 'A' THEN 'Accepted' WHEN STATUS = 'D' THEN 'Denied' WHEN STATUS IS NULL THEN 'Unknown' ELSE 'Other' END FROM CONTRACTS	

DECRYPT_BINARY, DECRYPT_BIT, DECRYPT_CHAR, and DECRYPT_DB

The decryption functions return a value that is the result of decrypting encrypted data. The decryption functions can decrypt only values that are encrypted by using the ENCRYPT_TDES function.



The schema is SYSIBM.

The password used for decryption is either the *password-string* value or the ENCRYPTION PASSWORD value, which is assigned by the SET ENCRYPTION PASSWORD statement.

encrypted-data
An expression that returns a complete, encrypted data value of a CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, BINARY, or VARBINARY data type. The data string must have been encrypted using the ENCRYPT_TDES function. The length attribute must be greater than or equal to 0 (zero) and less than or equal to 32672.

password-string
An expression that returns a CHAR or VARCHAR value with at least 6 bytes and no more than 127 bytes. This expression must be the same password that was used to encrypt the data or decryption will result in a different value than was originally encrypted. For enhanced security, *password-string* should be specified using a host variable rather than a string constant. If the value of the password argument is null or not provided, the data will be decrypted using the ENCRYPTION PASSWORD value, which must have been assigned by the SET ENCRYPTION PASSWORD statement.

For a static SQL statement, it is recommended that the password be specified with a host variable rather than with a string constant.

DEFAULT
The data is decrypted using the ENCRYPTION PASSWORD value, which must have been assigned by the SET ENCRYPTION PASSWORD statement.

ccsid-constant
A integer constant that specifies the CCSID in which the data should be returned by the decryption function. If DECRYPT_BIT or DECRYPT_BINARY is specified, *ccsid-constant* must not be specified. The default is

- The ENCODING bind option of the plan or package or the APPLICATION ENCODING SCHEMA option of the CREATE PROCEDURE or ALTER PROCEDURE statement for native SQL procedures that contain the static SQL statements
- The value of the APPLICATION ENCODING special register for dynamic SQL statements

The data type of the result of the function is determined by the function that is specified and the data type of the first argument, as shown in the following table. If the cast from the actual type of the encrypted data to the result of the function is not supported, a warning or error is returned.

Table 64. Result of the decryption function

Function	Type of first argument	Actual type of encrypted data	Result
DECRYPT_BINARY	FOR BIT DATA[1], BINARY, VARBINARY	Any string (except for LOBs)	VARBINARY
DECRYPT_BIT	FOR BIT DATA, BINARY, VARBINARY	CHAR, VARCHAR	VARCHAR FOR BIT DATA
DECRYPT_BIT	FOR BIT DATA, BINARY, VARBINARY	GRAPHIC, VARGRAPHIC (UTF16)	Warning or error If a warning is returned, the result is VARCHAR FOR BIT DATA
DECRYPT_BIT	FOR BIT DATA, BINARY, VARBINARY	GRAPHIC, VARGRAPHIC (not UTF16)	Warning or error If a warning is returned, the result is VARCHAR FOR BIT DATA
DECRYPT_BIT	FOR BIT DATA, BINARY, VARBINARY	BINARY, VARBINARY	Warning or error If a warning is returned, the result is VARCHAR FOR BIT DATA
DECRYPT_CHAR	FOR BIT DATA, BINARY, VARBINARY	CHAR, VARCHAR	VARCHAR(3)
DECRYPT_CHAR	FOR BIT DATA, BINARY, VARBINARY	GRAPHIC, VARGRAPHIC (UTF16)	VARCHAR(3)
DECRYPT_CHAR	FOR BIT DATA, BINARY, VARBINARY	GRAPHIC, VARGRAPHIC (not UTF16)	Warning or error If a warning is returned, the result is VARCHAR(3)
DECRYPT_CHAR	FOR BIT DATA, BINARY, VARBINARY	BINARY, VARBINARY	Warning or error If a warning is returned, the result is VARCHAR(3)
DECRYPT_DB	FOR BIT DATA, BINARY, VARBINARY	CHAR, VARCHAR, GRAPHIC, VARGRAPHIC	VARGRAPHIC
DECRYPT_DB	FOR BIT DATA, BINARY, VARBINARY	BINARY, VARBINARY	Warning or error If a warning is returned, the result is VARGRAPHIC

Table 64. Result of the decryption function (continued)

Function	Type of first argument	Actual type of encrypted data	Result
Note: [1] FOR BIT DATA means CHAR or VARCHAR FOR BIT DATA			

If *encrypted-data* included a hint, the hint is not returned by the function. The length attribute of the result is the length attribute of *encrypted-data* minus 8 bytes. The actual length of the value that is returned by the function will match the length of the original string that was encrypted. If *encrypted-data* includes bytes beyond the encrypted string, these bytes are not returned by the function.

Administration of encrypted data: The decryption functions can only decrypt data that was encrypted using the Triple DES encryption algorithm. Therefore, columns with encrypted data can only be used after replication if they were encrypted using the Triple DES encryption algorithm.

If the data is decrypted using a different CCSID than the originally encrypted value, it is possible that expansion might occur when converting the decrypted value to this CCSID. In such situations, the *encrypted-data* value must first be cast to a VARCHAR string with a larger number of bytes before performing the decryption functions.

The result can be null; if the first argument is null, the result is the null value.

For additional information about using the decryption functions, see "ENCRYPT_TDES" on page 464 and "GETHINT" on page 476.

Password protection: To prevent inadvertent access to the encryption password, do not specify *password-string* as a string constant in the source statement. Instead, use the ENCRYPTION PASSWORD special register or specify the password using a host variable.

Example 1: Set the ENCRYPTION PASSWORD value to 'Ben123' and use it as the password to insert a decrypted social security number into the table. Decrypt the value of the added social security number, using the ENCRYPTION PASSWORD value.

```
SET ENCRYPTION PASSWORD ='Ben123';
INSERT INTO EMP(SSN) VALUES ENCRYPT_TDES('289-46-8832');
SELECT DECRYPT_CHAR(SSN) FROM EMP;
```

This example returns the value '289-46-8832'.

Example 2: Decrypt the social security number that is inserted into the table. Instead of using the ENCRYPTION PASSWORD value, explicitly specify 'Ben123' as the encryption password.

```
SELECT DECRYPT_CHAR(SSN,'Ben123') FROM EMP;
```

This example returns the value '289-46-8832'.

Example 3: Insert a decrypted social security number into the table, explicitly specifying 'Ben123' as the password. Decrypt the data and have it converted to CCSID 1208.

```
SET ENCRYPTION PASSWORD ='Ben123';
INSERT INTO EMP(SSN) VALUES ENCRYPT_TDES('289-46-8832');
SELECT DECRYPT_CHAR(SSN) FROM EMP;
```

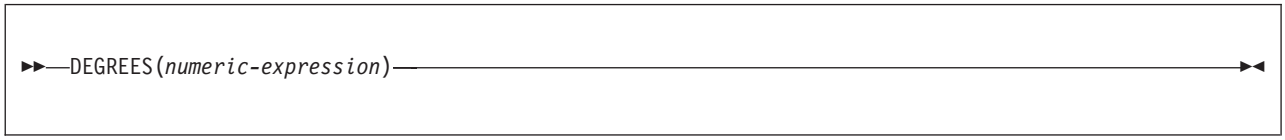
When a CCSID is specified, it might be necessary to explicitly cast the data to a longer value to ensure that there is room for expansion when the data is decrypted. The following example illustrates the technique:

```
SELECT DECRYPT_CHAR(CAST(SSN AS VARCHAR(57)),
                    'Ben123',1208)
               FROM EMP;
```

In the first case, where the data is not cast to a longer value, the result is a VARCHAR(11) value. In the second case, to allow for expansion, SSN is cast as `VARCHAR(57) (11 * 3 + 24)`. Casting the data to a longer value allows for three times expansion in the normal VARCHAR(11) result. Three times expansion is often associated with a worst case of ASCII or EBCDIC to Unicode UTF-8 conversion. In both cases in this example, the result is the VARCHAR(11) value '289-46-8832'.

DEGREES

The DEGREES function returns the number of degrees of the argument, which is an angle, expressed in radians.



The schema is SYSIBM.

The argument must be an expression that returns the value of any built-in numeric data type that is not DECFLOAT. If the argument is not a double precision floating-point number, it is converted to one for processing by the function.

The result of the function is a double precision floating-point number.

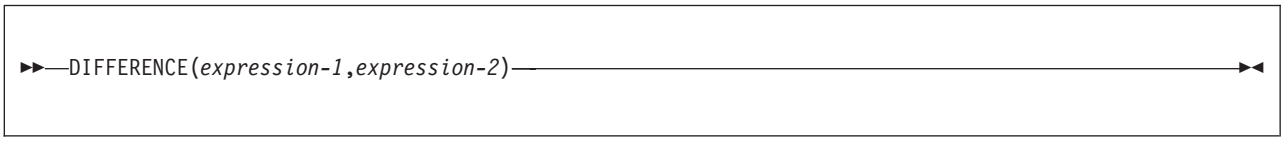
The result can be null; if the argument is null, the result is the null value.

Example: Assume that host variable HRAD is a DOUBLE with a value of 3.1415926536. The following statement returns a double precision floating-point number with an approximate value of 180.0.

```
SELECT DEGREES(:HRAD)
  FROM SYSIBM.SYSDUMMY1;
```

DIFFERENCE

The DIFFERENCE function returns a value, from 0 to 4, that represents the difference between the sounds of two strings, based on applying the SOUNDEX function to the strings. A value of 4 is the best possible sound match.



The schema is SYSIBM.

expression-1 **or** *expression-2*
Each expression must return a value that is a built-in numeric, character string, or graphic string data type that is not a LOB. A numeric argument is cast to a character string before the function is evaluated. For more information on converting a numeric string to a character string, see "VARCHAR" on page 673.

The data type of the result is INTEGER.

The result can be null; if any argument is null, the result is the null value.

Example 1: Find the DIFFERENCE and SOUNDEX values for 'CONSTRAINT' and 'CONSTANT':

```
SELECT DIFFERENCE('CONSTRAINT','CONSTANT'),
       SOUNDEX('CONSTRAINT'),
       SOUNDEX('CONSTANT')
  FROM SYSIBM.SYSDUMMY1;
```

This example returns the values 4, C523, and C523. Since the two strings return the same SOUNDEX value, the difference is 4 (the highest value possible).

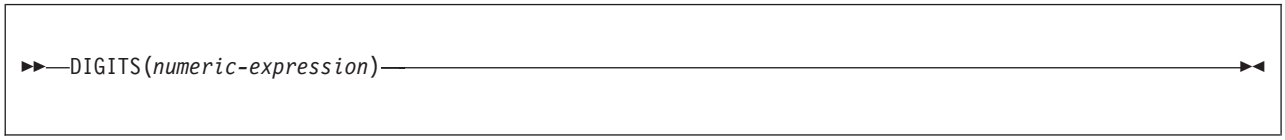
Example 2: Find the DIFFERENCE and SOUNDEX values for 'CONSTRAINT' and 'CONTRITE':

```
SELECT DIFFERENCE('CONSTRAINT','CONTRITE'),
       SOUNDEX('CONSTRAINT'),
       SOUNDEX('CONTRITE')
  FROM SYSIBM.SYSDUMMY1;
```

This example returns the values 2, C523, and C536. In this case, the two strings return different SOUNDEX values, and hence, a lower difference value.

DIGITS

The DIGITS function returns a character string representation of the absolute value of a number.



The schema is SYSIBM.

The argument must be an expression that returns a value that is a SMALLINT, INTEGER, BIGINT, or DECIMAL built-in numeric data type.

The result of the function is a fixed-length character string representing the absolute value of the argument without regard to its scale. The result does not include a sign or a decimal point. Instead, it consists exclusively of digits, including, if necessary, leading zeros to fill out the string. The length of the string is:

- 5 if the argument is a small integer
- 10 if the argument is a large integer
- 19 if the argument is a big integer
- *p* if the argument is a decimal number with a precision of *p*

The result can be null; if the argument is null, the result is the null value.

The CCSID of the result is determined from the context in which the function was invoked. For more information, see "Determining the encoding scheme and CCSID of a string" on page 47.

Example 1: Assume that an INTEGER column called INTCOL containing a 10-digit number is in a table called TABLEX. INTCOL has the data type INTEGER instead of CHAR(10) to save space. the following query lists all combinations of the first four digits in column INTCOL.

```
SELECT DISTINCT SUBSTR(DIGITS(INTCOL),1,4)
  FROM TABLEX;
```

Example 2: Assume that COLUMNX has the data type DECIMAL(6,2), and that one of its values is -6.28. For this value, the following statement returns the value '000628'.

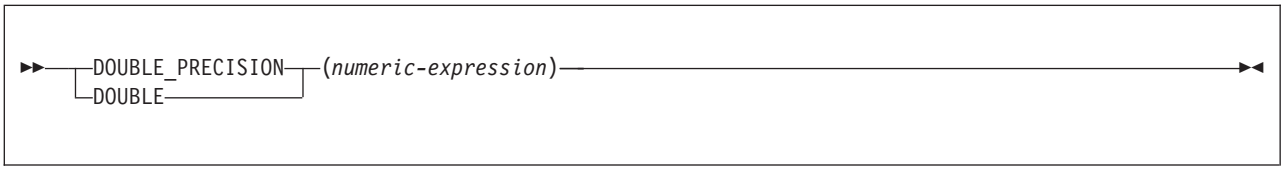
```
DIGITS(COLUMNX)
```

The result is a string of length six (the precision of the column) with leading zeros padding the string out to this length. Neither sign nor decimal point appear in the result.

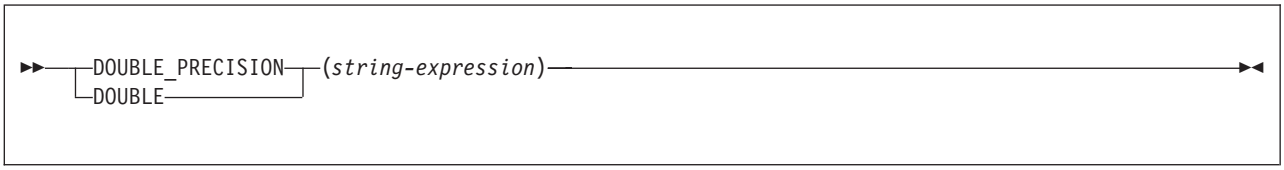
DOUBLE_PRECISION or DOUBLE

The DOUBLE_PRECISION and DOUBLE functions returns a floating-point representation of either a number or a character-string or graphic-string representation of a number, an integer, a decimal number, or a floating-point number.

Numeric to Double:



String to Double:



The schema is SYSIBM.

Numeric to Double

numeric-expression
An expression that returns a value of any built-in numeric data type.

The result is the same number that would occur if the expression were assigned to a double precision floating-point column or variable.

String to Double

string-expression
An expression that returns a value of a character or graphic string (except a CLOB or DBCLOB) with a length attribute that is not greater than 255 bytes. The string must contain a valid string representation of a number.

The result is the same number that would result from `CAST(`*string-expression*` AS DOUBLE PRECISION)`. Leading and trailing blanks are removed from the string, and the resulting substring must conform to the rules for forming a valid string representation of an SQL floating-point, integer, or decimal constant.

The result of the function is a double precision floating-point number.

The result can be null; if the argument is null, the result is the null value.

Note: To increase the portability of applications, use the CAST specification. For more information, see "CAST specification" on page 267.

FLOAT can be specified as a synonym for DOUBLE or DOUBLE_PRECISION.

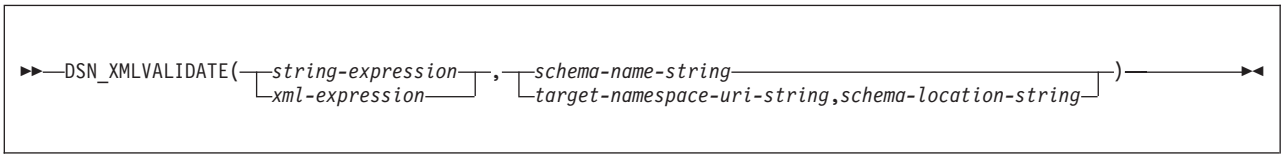
Example: Using sample table DSN8B10.EMP, find the ratio of salary to commission for employees whose commission is not zero. The columns involved in the

calculation, SALARY and COMM, have decimal data types. To eliminate the possibility of out-of-range results, apply the DOUBLE function to SALARY so that the division is carried out in floating-point.

```
SELECT EMPNO, DOUBLE(SALARY)/COMM
  FROM DSN8B10.EMP
  WHERE COMM > 0;
```

DSN_XMLVALIDATE

The DSN_XMLVALIDATE function returns an XML value that is the result of applying XML schema validation to the first argument of the function. DSN_XMLVALIDATE can validate XML data that has a maximum length of 2 GB - 1 byte.



The schema is SYSIBM.

string-expression
An expression that returns a built-in character, graphic, or binary string. The value must be a well-formed XML document that conforms to the XML Version 1.0 standard.

xml-expression
An expression that returns an XML value in the XML data type. The value must be a well-formed XML document that conforms to XML Version 1.0 standard.

schema-name-string
An expression that returns a built-in varying length character string that is not a CLOB. The value specifies the name of the XML schema object that is used for validation. The value must not be an empty string or the null value, and the actual length must be less than or equal to 257. If the XML schema name is qualified, the qualifier must be SYSXSR (SYSXSR is the default qualifier). The value must identify a registered XML schema in the DB2 XML schema repository.

target-namespace-uri-string
An expression that returns a built-in varying length character string that is not a CLOB, with a length attribute that is not greater than 1000. The value specifies the target namespace name or universal resource identifier (URI) of the XML schema that is to be used for validation. If the value is an empty string of the null value, no namespace is used to locate the XML schema.

schema-location-string
An expression that returns a built-in varying length character string that is not a CLOB, with a length attribute that is not greater than 1000. The value specifies the XML schema location hint URI of the XML schema that is to be used for validation. If the value is an empty string of the null value, no schema location is used to locate the XML schema.

If *target-namespace-uri-string* and *schema-location-string* are specified, the combination must identify a registered XML schema in the DB2 XML schema repository, and there must be only one such registered XML schema.

A schema must be registered successfully in the DB2 XML schema repository before it can be used for DSN_XMLVALIDATE. If the validation fails, DB2 returns an error.

The result of the function is an XML value.

The result can be null; if the first argument is null, the result is the null value.

Example 1: The following example shows how the DSN_XMLVALIDATE function validates the XML data that is contained in the *value_host_var* host variable. The XML schema, SYSXSR.ORDERSCHEMA, was registered prior to this statement:

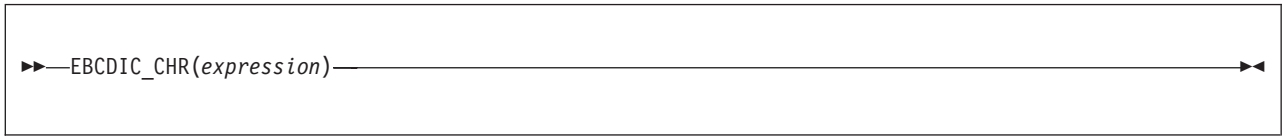
```
INSERT INTO T1(C1) VALUES(
 DSN_XMLVALIDATE(:value_host_var, 'SYSXSR.MYXMLSCHEMA'));
```

Example 2: The following example is similar to the previous example but references the namespace and schema location:

```
INSERT INTO T1(C1) VALUES(
 DSN_XMLVALIDATE(:value_host_var,
                       'http://www.n1.com',
                       'http://www.n1.com/report.xsd'));
```

EBCDIC_CHR

The EBCDIC_CHR function returns the character that has the EBCDIC code value that is specified by the argument.



The schema is SYSIBM.

expression
An expression that returns a BIGINT, INTEGER, or SMALLINT built-in data type value.

expression can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34) which is then assigned to a BIGINT value.

The result of the function is a CHAR(1) string encoded in the SBCS EBCDIC CCSID (regardless of the setting of the MIXED option in DSNHDECP). If the value of *expression* is not in the range of 0 to 255, the null value is returned.

The result can be null; if the argument is null, the result is the null value.

Example: Set *hv* with the Euro symbol "€" in CCSID 1140:

```
SET :hv = EBCDIC_CHR(159);  -- x'9F'
```

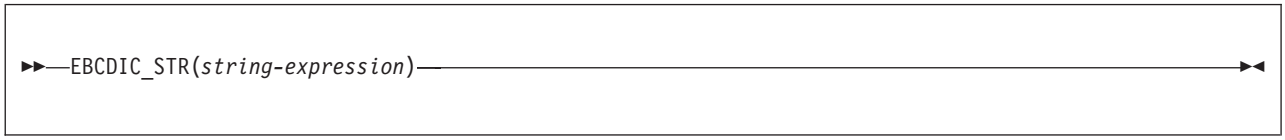
Set *hv* with the Euro symbol "€" in CCSID 1142:

```
SET :hv = EBCDIC_CHR(90);  -- x'5A'
```

In both cases, the "€" is assigned to *hv*, but because the Euro symbol is located at different code points for the two CCSIDs, the input value is different.

EBCDIC_STR

The EBCDIC_STR function returns a string, in the system EBCDIC CCSID, that is an EBCDIC version of the string.



The schema is SYSIBM.

The system EBCDIC CCSID is defined as the SBCS EBCDIC CCSID on a MIXED=NO system or the MIXED EBCDIC CCSID on a MIXED=YES system.

string-expression
An expression that returns a value of a built-in character or graphic string. If the string is a character string, in cannot be bit data. *string-expression* must be an ASCII, EBCDIC, or Unicode string. EBCDIC_STR returns an EBCDIC version of the string. Non-EBCDIC characters are converted to the form *\xxxx*, where *xxxx* represents a UTF-16 code unit.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.

The length attribute of the result is calculated using the formulas in Table 30 on page 142. The length attribute of the result will be `MAX( ( 5*n),32704)`. Where *n* is the result of applying the formulas in Table 30 on page 142 based on input and output data types.

The result of the function is an EBCDIC character string (in the system EBCDIC CCSID). If the actual length of the result string exceeds the maximum for the return type, an error occurs.

The result can be null; if the argument is null, the result is the null value.

Example: The following example returns the EBCDIC string equivalent of the text string "Hi my name is Андрей (Andrei)"

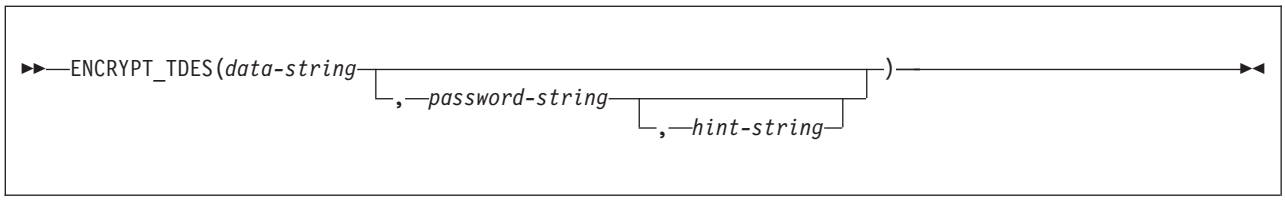
```
SET :HV1 = EBCDIC_STR('Hi, my name is Андрей (Andrei)');
```

HV1 is assigned the value "Hi, my name is \0410\043D\0434\0440\0435\0439 (Andrei)"

ENCRYPT_TDES

The ENCRYPT_TDES function returns a value that is the result of encrypting the first argument by using the Triple DES encryption algorithm. The function can also set the password that is used for encryption.

The encryption password can also be set by using the ENCRYPTION PASSWORD value, which is assigned by using the SET ENCRYPTION PASSWORD statement.



The schema is SYSIBM.

data-string
An expression that returns the string value to be encrypted. The string expression must return a built-in string data type that is not a LOB. The length attribute must be greater than or equal to 0 (zero). The length attribute is limited to 32640 if *hint-string* is specified and 32672 if *hint-string* is not specified.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.

password-string
An expression that returns a CHAR or VARCHAR value with at least 6 bytes and no more than 127 bytes.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.

The value represents the password that is used to encrypt *data-string*. If the value of the password argument is null or not specified, the data is encrypted using the ENCRYPTION PASSWORD value, which must have been assigned by the SET ENCRYPTION PASSWORD statement.

hint-string
An expression that returns a CHAR or VARCHAR value up to 32 bytes that is to help data owners remember passwords (for example, 'Ocean' as a hint to remember 'Pacific').

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.

If a hint value is specified, the hint is embedded into the result and can be retrieved using the GETHINT function. If this argument is null or not specified and no hint was specified when the ENCRYPTION PASSWORD was set, no hint is embedded in the result. If *password-string* is not specified, the hint can be specified using the SET ENCRYPTION PASSWORD statement.

The data type of the result is determined by the first argument as shown in the following table:

Table 65. Data type of the results of the ENCRYPT_TDES function

Data type of the first argument	Data type of the result
BINARY, VARBINARY	VARBINARY
CHAR, VARCHAR, GRAPHIC, VARGRAPHIC	VARCHAR FOR BIT DATA

The encoding scheme of the result is the same as the encoding scheme of *data-string*. If the result is character data, the result is bit data.

The length attribute of the result is different depending of whether *hint-string* is specified:

- If *hint-string* is specified, the length attribute of the result is the length attribute of the non-encrypted data + 24 bytes + number of bytes to the next 8 byte boundary + 32 bytes for the hint.
- If *hint-string* is not specified, the length attribute of the result is the length attribute of the non-encrypted data + 24 bytes + the number of bytes to the next 8 byte boundary.

The result can be null; if the first argument is null, the result is the null value.

The encrypted result is longer than the *data-string* value. Therefore, when assigning encrypted values, ensure that the target is declared with a length that can contain the entire encrypted value.

When encrypting data, be aware of the following points:

- **Password protection:** To prevent inadvertent access to the encryption password, do not specify *password-string* as a string constant in the source for a program, procedure, or function. Instead, use the SET ENCRYPTION PASSWORD statement or a variable.
- **Encryption algorithm:** The internal encryption algorithm used is Triple DES cipher block chaining (CBC) with padding. The 128-bit secret key is derived from the password using an MD5 hash.
- **Encryption passwords and data:** It is your responsibility to perform password management. After data is encrypted, only the password that is used to encrypt it can be used to decrypt it. If a different password is used to decrypt the data than was used to encrypt the data, the results of decryption will not match the original string. No error or warning is returned. CHAR variables might be padded with blanks if they are used to set password values. The encrypted result might contain null terminator and other non-printable characters.
- **Table column definitions:** When defining columns and types to contain encrypted data, always calculate the length attribute as follows:
  - For encrypted data with an embedded hint, the column length should be the length attribute of the non-encrypted data + 24 bytes + number of bytes to the next 8 byte boundary + 32 bytes for the hint.
  - For encrypted data without an embedded hint, the column length should be the length attribute of the non-encrypted data + 24 bytes + number of bytes to the next 8 byte boundary.

  Here are some sample column length calculations, which assume that a hint is not embedded:

```
Maximum length of non-encrypted data          6 bytes
24 bytes for encryption key                  24 bytes
Number of bytes to the next 8 byte boundary   2 bytes
```

```
                                                 ---------

  Encrypted data column length                    32 bytes
  Maximum length of non-encrypted data            32 bytes
  24 bytes for encryption key                     24 bytes
  Number of bytes to the next 8 byte boundary      0 bytes
                                                 ---------

  Encrypted data column length                    56 bytes
```

- **Administration of encrypted data:** Encrypted data can be decrypted only on servers that support the decryption of data that was encrypted using the Triple DES encryption algorithm. Hence, replication of columns with encrypted data should only be done to servers that support the decryption functions and the same encryption algorithms.

ENCRYPT can be specified as a synonym for ENCRYPT_TDES. DB2 supports this keyword to provide compatibility with other products in the DB2 family.

Example 1: Encrypt the social security number that is inserted into the table. Set the ENCRYPTION PASSWORD value to 'Ben123' and use it as the password.

```
SET ENCRYPTION PASSWORD ='Ben123';
INSERT INTO EMP(SSN) VALUES ENCRYPT_TDES ('289-46-8832');
```

Example 2: Encrypt the social security number that is inserted into the table. Explicitly specify 'Ben123' as the encryption password.

```
INSERT INTO EMP(SSN) VALUES ENCRYPT_TDES ('289-46-8832','Ben123');
```

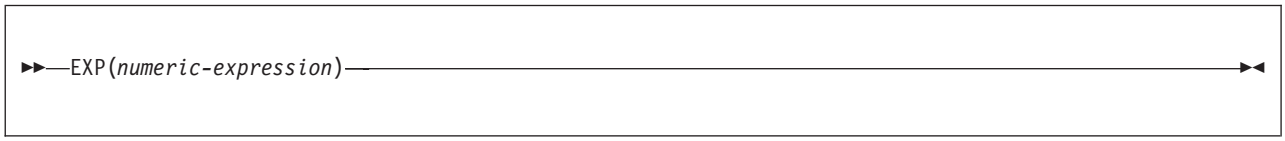
Example 3: Encrypt the social security number that is inserted into the table. Specify 'Pacific' as the encryption password, and provide 'Ocean' as a hint to help the user remember the password of 'Pacific'.

```
INSERT INTO EMP(SSN) VALUES ENCRYPT_TDES ('289-46-8832','Pacific','Ocean');
```

The preceding statement returns a double precision floating-point number with an approximate value of 31.62.

EXP

The EXP function returns a value that is the base of the natural logarithm (e), raised to a power that is specified by the argument. The EXP and LN functions are inverse operations.



The schema is SYSIBM.

The argument must be an expression that returns the value of any built-in numeric data type that is not DECFLOAT. If the argument is not a double precision floating-point number, it is converted to one for processing by the function.

The result of the function is a double precision floating-point number.

The result can be null; if the argument is null, the result is the null value.

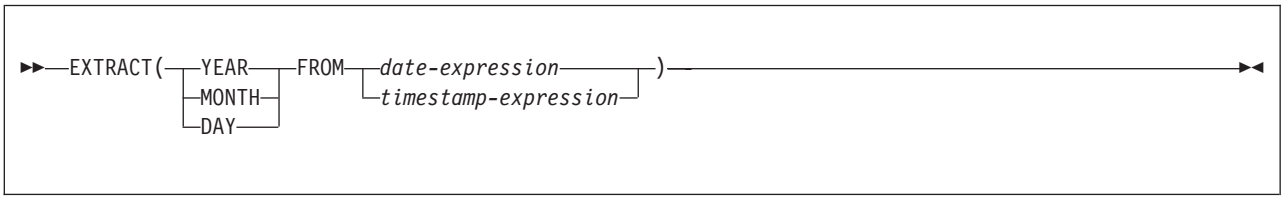
Example: Assume that host variable E is DECIMAL(10,9) with a value of 3.453789832. The following statement returns a double precision floating-point number with an approximate value of 31.62.

```
SELECT EXP(:E)
  FROM SYSIBM.SYSDUMMY1;
```

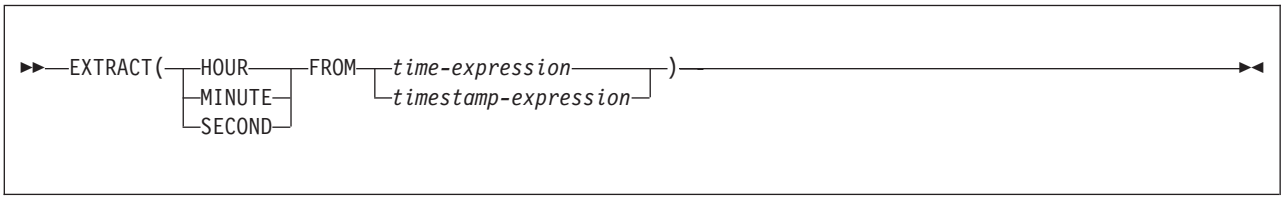
EXTRACT

The EXTRACT function returns a portion of a date or timestamp, based on its arguments.

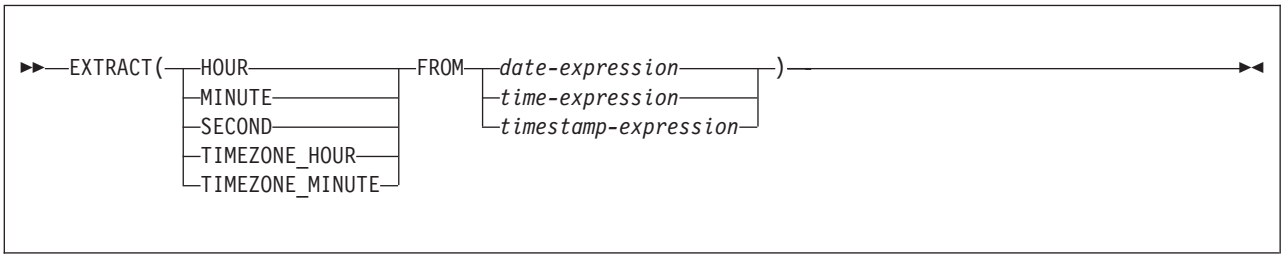
Extract date values:



Extract time values:



Extract time zone values:



The schema is SYSIBM.

The result can be null; if the argument is null, the result is the null value.

Extract date values

YEAR
Specifies that the year portion of *date-expression* or *timestamp-expression* is returned. The result is identical to the YEAR scalar function. For more information, see "YEAR" on page 732.

MONTH
Specifies that the month portion of *date-expression* or *timestamp-expression* is returned. The result is identical to the MONTH scalar function. For more information, see "MONTH" on page 535.

DAY
Specifies that the day portion of *date-expression* or *timestamp-expression* is returned. The result is identical to the DAY scalar function. For more information, see "DAY" on page 427.

date-expression
An expression that returns the value of either a built-in date or built-in character string data type.

If *date-expression* is a character or graphic string, it must not be a CLOB or DBCLOB and its value must be a valid character-string or graphic-string representation of a date. For the valid formats of string representations of dates, see "String representations of datetime values" on page 101.

timestamp-expression
An expression that returns the value of either a built-in timestamp or built-in character string data type.

If *timestamp-expression* is a character or graphic string, it must not be a CLOB or DBCLOB and its value must be a valid character-string or graphic-string representation of a timestamp. For the valid formats of string representations of timestamps, see "String representations of datetime values" on page 101.

Extract time values

HOUR
Specifies that the hour portion of *time-expression* or *timestamp-expression* is returned. The result is identical to the HOUR scalar function. For more information, see "HOUR" on page 484.

MINUTE
Specifies that the minute portion of *time-expression* or *timestamp-expression* is returned. The result is identical to the MINUTE scalar function. For more information, see "MINUTE" on page 531.

SECOND
Specifies that the second portion of *time-expression* or *timestamp-expression* is returned. The result is identical to the SECOND scalar function where the precision and scale of the result depend on the type of *time-expression* or *timestamp-expression*. For more information, see "SECOND" on page 603.

time-expression
An expression that returns the value of either a built-in time or built-in character string data type.

If *time-expression* is a character or graphic string, it must not be a CLOB or DBCLOB and its value must be a valid string representation of a time. For the valid formats of string representations of times, see "String representations of datetime values" on page 101.

timestamp-expression
An expression that returns the value of either a built-in timestamp or built-in character string data type.

If *timestamp-expression* is a character or graphic string, it must not be a CLOB or DBCLOB and its value must be a valid string representation of a timestamp. For the valid formats of string representations of timestamps, see "String representations of datetime values" on page 101.

Extract time zone values

TIMEZONE_HOUR
Specifies that the hour component of the time zone of the timestamp

value is returned. TIMEZONE_HOUR can only be specified if the second argument is a *timestamp-expression* and the *timestamp-expression* contains a time zone.

TIMEZONE_MINUTE
Specifies that the minute component of the time zone of the timestamp value is returned. TIMEZONE_MINUTE can only be specified if the second argument is a *timestamp-expression* and the *timestamp-expression* contains a time zone.

The values of **TIMEZONE_HOUR** and **TIMEZONE_MINUTE** shall either both be non-negative or both be non-positive.

If the *timestamp-expression* argument includes a time zone, the result is determined from the UTC representation of the datetime value.

The data type of the result of the function depends on the part of the datetime value that is specified:

- The result is INTEGER, if one of the following is specified:
  - YEAR
  - MONTH
  - DAY
  - HOUR
  - MINUTE
  - TIMEZONE_HOUR
  - TIMEZONE_MINUTE
- The result is DECIMAL(2+*p*, *p*) where *p* is the fractional second precision, if SECOND is specified with a TIMESTAMP(*p*) value.
- The result is DECIMAL(8,6), if SECOND is specified with a TIME value or a string representation of a TIME or timestamp. The fractional digits contains fractional seconds.

Example 1:
Assume that the column PRSTDATE has an internal value that is equivalent to 2010-12-25. The following statement returns the value 12:

```
SELECT EXTRACT(MONTH FROM PRSTDATE)
  FROM PROJECT;
```

Example 2:
Assume that host variable *PRSTSZ* contains the value 2008-02-29.20.00.000000 -08.30:

```
SELECT EXTRACT(HOUR FROM :PRSTSZ) FROM PROJECT;
```

The SELECT statement returns the value 4, which is the hour of the input datetime value expressed in UTC.

To return the same hour value as expressed in the input, cast the value to TIMESTAMP WITHOUT TIME ZONE before using the EXTRACT function:

```
SELECT EXTRACT(HOUR FROM CAST (:PRSTSZ AS TIMESTAMP ) ) FROM PROJECT;
```

The SELECT statement returns the value 20, which is the hour as it was originally expressed as a string in the host variable.

```
SELECT EXTRACT(TIMEZONE_HOUR FROM :PRSTSZ) FROM PROJECT;
```

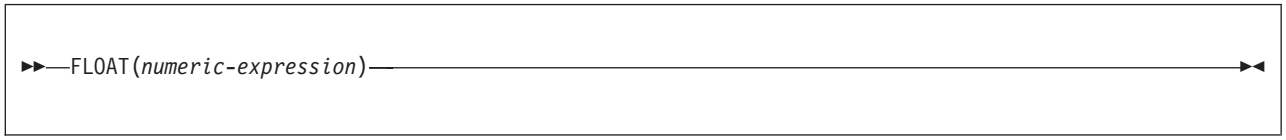
This SELECT statement returns the value -8.

```
SELECT EXTRACT(TIMEZONE_MINUTE FROM :PRSTSZ) FROM PROJECT;
```

This SELECT statement returns the value -30.

FLOAT

The FLOAT function returns a floating-point representation of either a number or a string representation of a number. FLOAT is a synonym for the DOUBLE function.

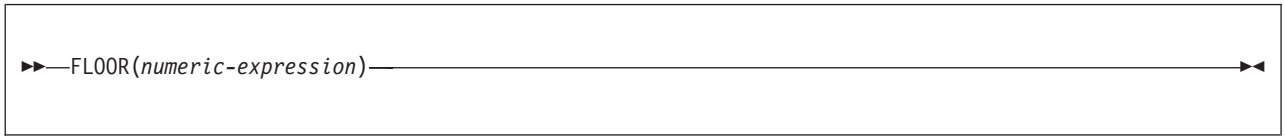


The schema is SYSIBM.

FLOAT is a synonym for the DOUBLE function. See “DOUBLE_PRECISION or DOUBLE” on page 458 for details.

FLOOR

The FLOOR function returns the largest integer value that is less than or equal to the argument.



The schema is SYSIBM.

The argument must be an expression that returns a value of any built-in numeric data type.

The argument can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34).

The result of the function has the same data type and length attribute as the argument. When the argument is DECIMAL, the scale of the result is 0 and not the scale of the input argument. For example, an argument with a date type of DECIMAL(5,5) results in DECIMAL(5,0).

The result can be null; if the argument is null, the result is the null value.

Example 1: Using sample table DSN8B10.EMP, find the highest monthly salary, rounding the result down to the next integer. The SALARY column has a decimal data type.

```
SELECT FLOOR(MAX(SALARY)/12)
  FROM DSN8B10.EMP;
```

This example returns 04395 because the highest paid employee is Christine Haas who earns $52750.00 per year. Her average monthly salary before applying the FLOOR function is 4395.83.

Example 2: This example demonstrates using FLOOR with both positive and negative numbers.

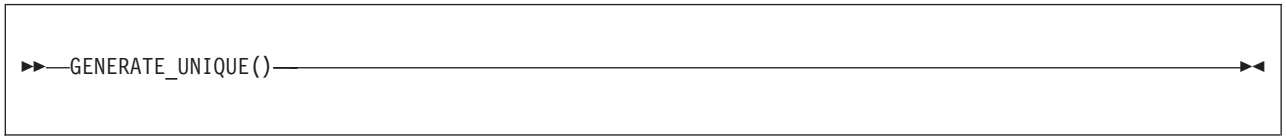
```
SELECT FLOOR( 3.5),
       FLOOR( 3.1),
       FLOOR(-3.1),
       FLOOR(-3.5)
FROM SYSIBM.SYSDUMMY1;
```

This example returns (leading zeros are shown to demonstrate the precision and scale of the result):

```
03. 03. -04. -04.
```

GENERATE_UNIQUE

The GENERATE_UNIQUE function returns a bit data character string that is unique, compared to any other execution of the same function.



The schema is SYSIBM.

The GENERATE_UNIQUE function returns a bit data character string 13 bytes long (CHAR(13) FOR BIT DATA) that is unique compared to any other execution of the same function. The function is defined as not deterministic. Although the function has no arguments, the empty parentheses must be specified when the function is invoked.

The result of the function is a unique value that includes the internal form of the Universal Time, Coordinated (UTC) and, if in a sysplex environment, the sysplex member where the function was processed. The result cannot be null.

The result of this function can be used to provide unique values in a table. Each successive value will be greater than the previous value, providing a sequence that can be used within a table. The sequence is based on the time when the function was executed.

This function differs from using the special register CURRENT TIMESTAMP in that a unique value is generated for each row of a multiple row insert statement, an insert statement with a fullselect, or an insert operation in a MERGE statement.

The timestamp value that is part of the result of this function can be determined using the TIMESTAMP function with the result of GENERATE_UNIQUE as an argument.

Example: Create a table that includes a column that is unique for each row. Populate this column using the GENERATE_UNIQUE function. Notice that the UNIQUE_ID column is defined as FOR BIT DATA to identify the column as a bit data character string.

```
CREATE TABLE EMP_UPDATE
 (UNIQUE_ID VARCHAR(13)FOR BIT DATA,
  EMPNO CHAR(6),
  TEXT VARCHAR(1000));
INSERT INTO EMP_UPDATE VALUES (GENERATE_UNIQUE(),'000020','Update entry 1...');
INSERT INTO EMP_UPDATE VALUES (GENERATE_UNIQUE(),'000050','Update entry 2...');
```

This table will have a unique identifier for each row if GENERATE_UNIQUE is always used to set the value the UNIQUE_ID column. You can create an insert trigger on the table to ensure that GENERATE_UNIQUE is used to set the value:

```
CREATE TRIGGER EMP_UPDATE_UNIQUE
  NO CASCADE BEFORE INSERT ON EMP_UPDATE
  REFERENCING NEW AS NEW_UPD
  FOR EACH ROW MODE DB2SQL
  SET NEW_UPD.UNIQUE_ID = GENERATE_UNIQUE();
```

With this trigger, the previous INSERT statements that were used to populate the table could be issued without specifying a value for the UNIQUE_ID column:

```
INSERT INTO EMP_UPDATE (EMPNO,TEXT) VALUES ('000020','Update entry 1...');
INSERT INTO EMP_UPDATE (EMPNO,TEXT) VALUES ('000050','Update entry 2...');
```

The timestamp (in UTC) for when a row was added to EMP_UPDATE can be returned using:

```
SELECT TIMESTAMP(UNIQUE_ID), EMPNO, TEXT FROM EMP_UPDATE;
```

Therefore, the table does not need a timestamp column to record when a row is inserted.

GETHINT

The GETHINT function returns a hint for the password if a hint was embedded in the encrypted data. A password hint is a phrase that helps you remember the password with which the data was encrypted. For example, 'Ocean' might be used as a hint to help remember the password 'Pacific'.

The schema is SYSIBM.

encrypted-data
An expression that returns a string that contains a complete, encrypted data string. *encrypted-data* must return a value that is a CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, BINARY, or VARBINARY built-in data type. The string must have been encrypted using ENCRYPT_TDES function.

The result of the function is VARCHAR(32). The actual length of the result is the actual length of the hint that was provided when the data was encrypted.

The result can be null; if the argument is null, the result is the null value.

If no hint was specified when the ENCRYPT_TDES function was used to encrypt the data, the result is the null value.

The encoding scheme of the result is the same as the encoding scheme of *encrypted-data*. If *encrypted-data* is bit data, the CCSID of the result is the default character CCSID for that encoding scheme. Otherwise, the CCSID of the result is the same as the CCSID of *encrypted-data*.

For additional information about this function, see “DECRYPT_BINARY, DECRYPT_BIT, DECRYPT_CHAR, and DECRYPT_DB” on page 451 and “ENCRYPT_TDES” on page 464.

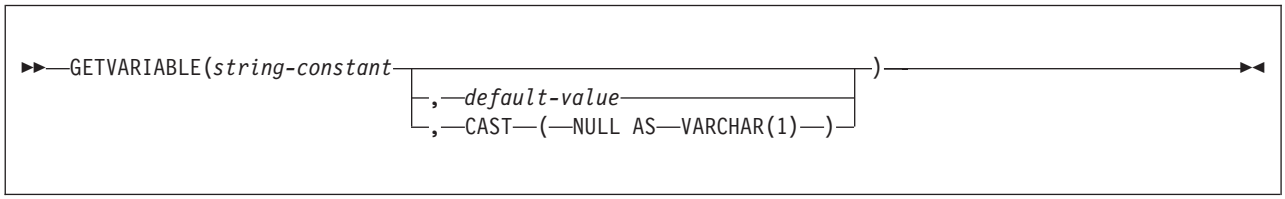
Example: This example shows how to embed a hint for the password when encrypting data and how to later use the GETHINT function to retrieve the embedded hint. In this example, the hint 'Ocean' is used to help remember the encryption password 'Pacific'.

```
INSERT INTO EMP (SSN) VALUES ENCRYPT_TDES ('289-46-8832','Pacific','Ocean');
SELECT GETHINT (SSN) FROM EMP;
```

The value that is returned is 'Ocean'.

GETVARIABLE

The GETVARIABLE function returns a varying-length character-string representation of the current value of the session variable that is identified by the argument.



The schema is SYSIBM.

string-constant
Specifies a string constant that contains the name of the session variable whose value is to be returned. The string constant:

- Must have a length that does not exceed 142 bytes.
- Must contain the fully qualified name of the variable, with no embedded blanks. Delimited identifiers must not be specified.
- Must not contain lowercase letters or characters that cannot be specified in an ordinary identifier.

The schema qualifier for the variable must be:

- SYSIBM for global variables. For a list of the built-in session variables, see "References to built-in session variables" on page 225.
- SESSION for user-defined session variables. User-defined session variables are established via the connection or signon exit routines.

Note: The GETVARIABLE function can obtain the values of only these session variables. This function cannot obtain the values of built-in global variables or user-defined global variables.

default-value
Specifies a string constant that contains the value to be returned if the specified variable does not exist or is not supported by DB2. *default-value* must be a string constant that does not exceed 255 bytes.

If *default-value* is not specified and the specified user-defined session variable does not exist or the built-in session variable is not supported by DB2, an error is returned.

CAST(NULL AS VARCHAR(1))
Specifies that a null value is to be returned if the specified variable does not exist or is not supported by DB2.

The data type of the result is VARCHAR(255). The result can be null.

The CCSID of the result is the CCSID for Unicode mixed data.

Example 1: Use the GETVARIABLE function to set the value of host variable :hv1 to the name of the plan that is currently being executed. The name of the built-in session variable that contains the name of the plan is SYSIBM.PLAN_NAME.

```
SET :hv1 = GETVARIABLE('SYSIBM.PLAN_NAME');
```

If DB2 does not support the name of the session variable, an error is returned. For example, the following statement returns an error because DB2 does not support a built-in session variable that is named SYSIBM.XYZ.

```
SET :hv1 = GETVARIABLE('SYSIBM.XYZ');
```

Example 2: Use the GETVARIABLE function to set the value of host variable :hv2 to the value for the user that is defined in user-defined session variable TEST. If the session variable has not been set or cannot be found, have the function return the value 'TEST FAILED'.

```
SET :hv2 = GETVARIABLE('SESSION.TEST','TEST FAILED');
```

Example 3: Use the GETVARIABLE function to set the value of host variable :hv3 to a string representation of the SYSTEM EBCDIC CCSIDs. The name of the built-in session variable that contains the system EBCDIC CCSIDs is SYSIBM.SYSTEM_EBCDIC_CCSID.

```
SET :hv3 = GETVARIABLE('SYSIBM.SYSTEM_EBCDIC_CCSID');
```

Regardless of the setting of the field MIXED DATA on the installation panel (YES or NO), the function returns three comma-delimited values that correspond to the SBCS, MIXED, and GRAPHIC CCSIDs for the encoding scheme.

For example, if the statement were issued on a system with the field MIXED DATA on the installation panel equal to NO and the default system CCSID of 37, this string would be returned:

'37,65534,65534'

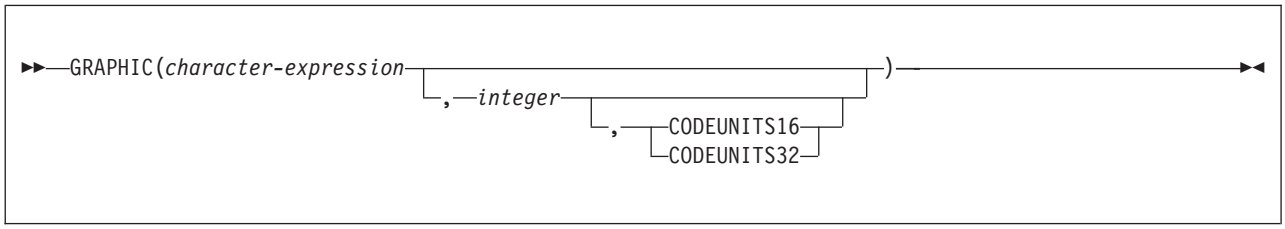
If the statement were issued on a system with the field MIXED DATA on the installation panel equal to YES and a default system CCSID of 930 (the mixed CCSID for the system), this string would be returned:

'290,930,300'

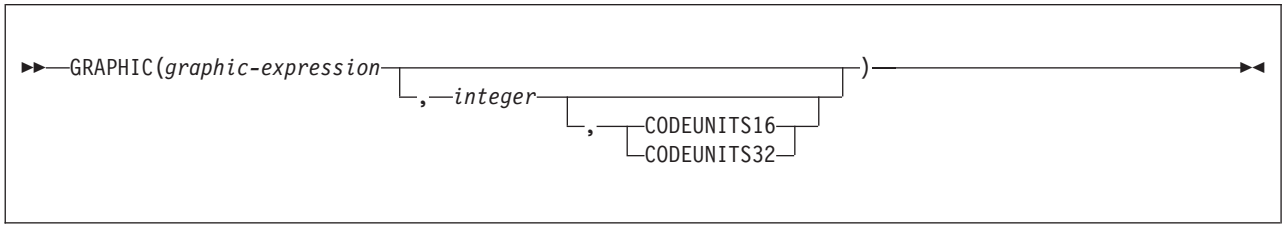
GRAPHIC

The GRAPHIC function returns a fixed-length graphic-string representation of a character string or a graphic string value, depending on the type of the first argument.

Character to Graphic:



Graphic to Graphic:



The schema is SYSIBM.

The result of the function is a fixed-length graphic string (GRAPHIC).

The result can be null; if the argument is null, the result is the null value.

The length attribute of the result is measured in double-byte characters because it is a graphic string.

Character to Graphic

character-expression
An expression that returns a value that is an EBCDIC-encoded or Unicode-encoded character string. It cannot be BIT data. The argument does not need to be mixed data, but any occurrences of X'0E' and X'0F' in the string must conform to the rules for EBCDIC mixed data. (See "Character strings" on page 84 for these rules.)

The value of the expression must not be an empty string if *integer* is not specified or have the value X'0E0F' if the string is an EBCDIC string.

integer
The length of the resulting fixed-length graphic string in the units that are either implicitly or explicitly specified. The value must be an integer constant between 1 and 127. If the length of *character-expression* is less than the length specified, the result is padded with double-byte blanks to the length of the result.

If CODEUNITS16 or CODEUNITS32 is specified, see "Determining the length attribute of the final result" on page 90 for information about how to calculate the length attribute of the result string.

If *integer* is not specified, the length of the result for an EBCDIC string is the minimum of 127 and the length attribute of *character-expression*, excluding shift characters. For a Unicode (UTF-8) string, the length is data dependent, but does not exceed 127.

CODEUNITS16 or CODEUNITS32
Specifies the unit that is used to express *integer*. If CODEUNITS16 or CODEUNITS32 is specified, the input is EBCDIC, and there is no system CCSID for EBCDIC GRAPHIC data, an error occurs.

CODEUNITS16
Specifies that *integer* is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32
Specifies that *integer* is expressed in terms of 32-bit UTF-32 code units.

For more information about CODEUNITS16 and CODEUNITS32, see "String unit specifications" on page 87.

The CCSID of the result is the graphic CCSID that corresponds to the character CCSID of *character-expression*. If the input is EBCDIC and there is no system CCSID for EBCDIC GRAPHIC data, the CCSID of the result is X'FFFE'.

For EBCDIC data, each character of *character-expression* determines a character of the result. The argument might need to be converted to the native form of mixed data before the result is derived. Let *M* be the system CCSID for mixed data. The argument is not converted if any of the following conditions is true:

- The argument is mixed data and its CCSID is *M*.
- The argument is SBCS data and its CCSID is the same as the system CCSID for SBCS data. In this case, the operation proceeds as if the CCSID of the argument is *M*.

Otherwise, the argument is a new string *S* derived by converting the characters to the coded character set identified by *M*. If there is no system CCSID for EBCDIC mixed data, conversion is to the coded character set that the system CCSID for SBCS data identifies.

The result is derived from *S* using the following steps:
- Each shift character (X'0E' or X'0F') is removed.
- Each double-byte character remains as is.
- Each single-byte character is replaced by a double-byte character.

The replacement for an SBCS character is the equivalent DBCS character if an equivalent exists. Otherwise, the replacement is X'FEFE'. The existence of an equivalent character depends on *M*. If there is no system CCSID for mixed data, the DBCS equivalent of X'xxxx' for EBCDIC is X'42xx', except for X'40', whose DBCS equivalent is X'4040'.

For Unicode data:

Each character of *character-expression* determines a character of the result. The argument might need to be converted to the native form of mixed data before the result is derived. Let M be the system CCSID for mixed data. The argument is not converted if any of the following conditions is true:

- The argument is mixed data, and its CCSID is *M*.

- The argument is SBCS data, and its CCSID is the same as the system CCSID for SBCS data. In this case, the operation proceeds as if the CCSID of the argument is *M*.

Otherwise, the argument is a new string *S* derived by converting the characters to the coded character set identified by *M*.

The result is derived from *S* by using the following steps:
- Each non-supplementary character is replaced by a Unicode double-byte character (a UTF-16 code point). A non-supplementary character in UTF-8 is between 1 and 3 bytes.
- Each supplementary character is replaced by a pair of Unicode double-byte characters (a pair of UTF-16 code points).

The replacement for a single-byte character is the Unicode equivalent character if an equivalent exists. Otherwise, the replacement is X'FEFE'.

Graphic to Graphic

graphic-expression
An expression that returns a value that is a graphic string. The graphic string must not be an empty string if *integer* is not specified.

integer
The length of the resulting fixed-length graphic string in the units that are either implicitly or explicitly specified. The value must be an integer constant between 1 and 127. If the length of *graphic-expression* is less than the length specified, the result is padded with double-byte blanks to the length of the result.

If CODEUNITS16 or CODEUNITS32 is specified, see "Determining the length attribute of the final result" on page 90 for information about how to calculate the length attribute of the result string.

If *integer* is not specified, the length of the result is the minimum of 127 and the length attribute of *graphic-expression*.

CODEUNITS16 or CODEUNITS32
Specifies the unit that is used to express *integer*. If CODEUNITS16 or CODEUNITS32 is specified, the input is EBCDIC, and there is no system CCSID for EBCDIC GRAPHIC data, an error occurs.

CODEUNITS16
Specifies that *integer* is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32
Specifies that *integer* is expressed in terms of 32-bit UTF-32 code units.

For more information about CODEUNITS16 and CODEUNITS32, see "String unit specifications" on page 87.

If the length of the *graphic-expression* is greater than the specified length of the result, the result is truncated. Unless all the truncated characters are blanks, a warning is returned.

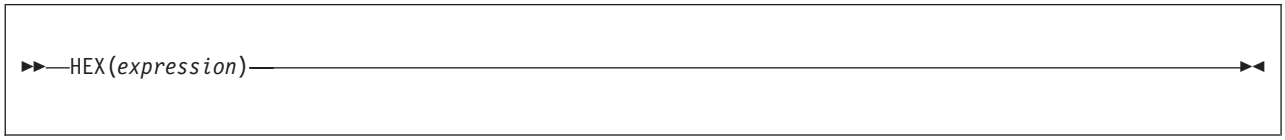
The CCSID of the result is the same as the CCSID of *graphic-expression*.

Example: Assume that MYCOL is a VARCHAR column in TABLEY. The following function returns the string in MYCOL as a fixed-length graphic string.

```
SELECT GRAPHIC(MYCOL)
  FROM TABLEY;
```

HEX

The HEX function returns a hexadecimal representation of a value.



The schema is SYSIBM.

The argument must an expression that returns a value of any built-in data type that is not XML. A character or binary string must not have a maximum length greater than 16352. A graphic string must not have a maximum length greater than 8176.

The result of the function is a character string.

The result can be null; if the argument is null, the result is the null value.

The result is a string of hexadecimal digits. The first two represent the first byte of the argument, the next two represent the second byte of the argument, and so forth. If the argument is a datetime value, the result is the hexadecimal representation of the internal form of the argument.

If the argument is a fixed-length string and the length of the result is less than 255, the result is a fixed-length string. Otherwise, the result is a varying-length string with a length attribute that depends on the following considerations:

If the argument is not a varying-length string, the length attribute of the result string is the same as the length of the result.

If the argument is a varying-length character or binary string, the length attribute of the result string is twice the length attribute of the argument.

If the argument is a varying-length graphic string, the length attribute of the result string is four times the length attribute of the argument.

If *expression* returns string data, the CCSID of the result is the SBCS CCSID that corresponds to the CCSID of *expression*. Otherwise, the CCSID of the result is determined from the context in which the function was invoked. For more information, see "Determining the encoding scheme and CCSID of a string" on page 47.

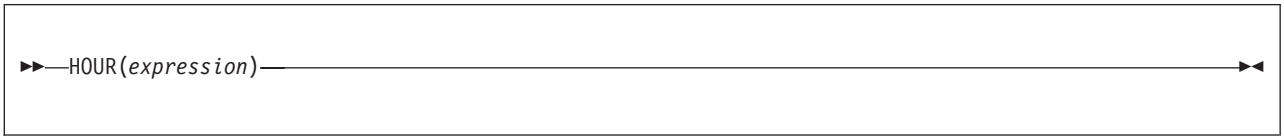
If the argument is a graphic string, the length of the result is four times the maximum length of the argument. Otherwise, the length of the result is twice the (maximum) length of the argument.

Example: Return the hexadecimal representation of START_RBA in the SYSIBM.SYSCOPY catalog table.

```
SELECT HEX(START_RBA) FROM SYSIBM.SYSCOPY;
```

HOUR

The HOUR function returns the hour part of a value.



The schema is SYSIBM.

The argument must be an expression that returns a value of one of the following built-in data types: a time, a timestamp, a character string, a graphic string, or a numeric data type.

- If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a time or timestamp with an actual length of not greater than 255 bytes. For the valid formats of string representations of times and timestamps, see "String representations of datetime values" on page 101.
- If *expression* is a number, it must be a time or timestamp duration. For the valid formats of time and timestamp durations, see "Datetime operands" on page 147.

If *expression* is a timestamp with a time zone, or a valid string representation of a timestamp with a time zone, the result is determined from the UTC representation of the datetime value.

The result of the function is a large integer.

The result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

If the argument is a time, timestamp, or string representation of either, the result is the hour part of the value, which is an integer between 1 and 24.

If the argument is a time duration or timestamp duration, the result is the hour part of the value, which is an integer between -99 and +99. A nonzero result has the same sign as the argument.

If the argument contains a time zone, the result is the year part of the value expressed in UTC.

Example 1: Assume that a table named CLASSES contains a row for each scheduled class. Also assume that the class starting times are in a TIME column named STARTTM. Select those rows in CLASSES that represent classes that start after the noon hour.

```
SELECT *
  FROM CLASSES
  WHERE HOUR(STARTTM) > 12;
```

Example 2: The following invocations of the HOUR function returns the same result:

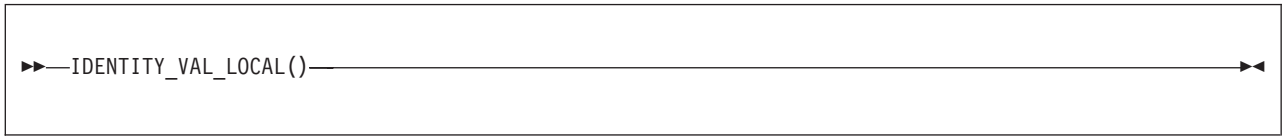
```
SELECT HOUR('2003-01-02-20.00.00'),
   HOUR('2003-01-02-12.00.00-08:00'),
   HOUR('2003-01-03-05.00.00+09:00')
  FROM SYSIBM.SYSDUMMY1;
```

For each invocation of the HOUR function in this SELECT statement, the result is 20.

When the input argument contains a time zone, the result is determined from the UTC representation of the input value. The string representations of a timestamp with a time zone in the SELECT statement all have the same UTC representation: 2003-01-02-20.00.00.

IDENTITY_VAL_LOCAL

The IDENTITY_VAL_LOCAL function returns the most recently assigned value for an identity column.



The schema is SYSIBM.

The IDENTITY_VAL_LOCAL function is not deterministic.[20] Although the function has no input parameters, the empty parentheses must be specified when the function is invoked.

The result is DECIMAL(31,0), regardless of the actual data type of the identity column to which the result value corresponds.

A *qualifying data change statement* refers to an insert operation (specified in either an INSERT statement or a MERGE statement).

The value that is returned is the value that was assigned to the identity column of the table identified in the most recent qualifying data change statement or LOAD utility operation for a table with an identity column. The insert operation has to be issued at the same level; that is, the value has to be available locally within the level at which it was assigned until replaced by the next assigned value. A new level is initiated when a trigger, function, or stored procedure is invoked. A trigger condition is at the same level as the associated triggered action.

The assigned value can be a value supplied by the user (if the identity column is defined as GENERATED BY DEFAULT) or an identity value that was generated by DB2.

Note: Use a SELECT FROM data change statement to obtain the assigned value for an identity column. See data-change-table-reference for more information.

The result can be null. The result is null in the following situations:

- When a qualifying data change statement has not been issued for a table containing an identity column at the current processing level
- When a COMMIT or ROLLBACK of a unit of work occurred since the most recent qualifying data change statement that assigned a value

The result of the function is not affected by a ROLLBACK TO SAVEPOINT statement.

Invoking the function within a qualifying data change statement: Expressions in a qualifying data change statement are evaluated before values are assigned to the target columns of the qualifying data change statement. Thus, when you invoke IDENTITY_VAL_LOCAL in a qualifying data change statement, the value that is

20. Being not deterministic affects what optimization (such as view processing and parallel processing) can be done when this function is used and in what contexts the function can be invoked. For example, the RAND function is another built-in scalar function that is not deterministic. Using functions that are not deterministic within a predicate can cause unpredictable results.

used is the most recently assigned value for an identity column from a previous qualifying data change statement. The function returns the null value if no such qualifying data change statement had been executed within the same level as the invocation of the IDENTITY_VAL_LOCAL function. Each qualifying data change statement that involves an IDENTITY column causes the identity value to be copied into connection-specific storage in DB2. Thus, the most recent identity value is used for a connection, regardless of what is happening with other concurrent user connections.

Invoking the function following a failed insert operation: The function returns an unpredictable result when it is invoked after the unsuccessful execution of a qualifying data change statement for a table with an identity column. The value might be the value that would have been returned from the function had it been invoked before the failed qualifying data change statement or the value that would have been assigned had the qualifying data change statement succeeded. The actual value returned depends on the point of failure and is therefore unpredictable.

Invoking the function within the SELECT statement of a cursor: Because the results of the IDENTITY_VAL_LOCAL function are not deterministic, the result of an invocation of the IDENTITY_VAL_LOCAL function from within the SELECT statement of a cursor can vary for each FETCH statement.

Invoking the function within the trigger condition of an insert trigger: The result of invoking the IDENTITY_VAL_LOCAL function from within the condition of an insert trigger is the null value.

Invoking the function within a triggered action of an insert trigger: Multiple before or after insert triggers can exist for a table. In such cases, each trigger is processed separately, and identity values generated by SQL statements issued within a triggered action are not available to other triggered actions using the IDENTITY_VAL_LOCAL function. This is the case even though the multiple triggered actions are conceptually defined at the same level.

Do not use the IDENTITY_VAL_LOCAL function in the triggered action of a before insert trigger. The result of invoking the IDENTITY_VAL_LOCAL function from within the triggered action of a before insert trigger is the null value.

The value for the identity column of the table for which the trigger is defined cannot be obtained by invoking the IDENTITY_VAL_LOCAL function within the triggered action of a before insert trigger. However, the value for the identity column can be obtained in the triggered action by referencing the trigger transition variable for the identity column.

The result of invoking the IDENTITY_VAL_LOCAL function in the triggered action of an after insert trigger is the value assigned to an identity column of the table identified in the most recent qualifying data change statement. That statement is the one invoked in the same triggered action that had a qualifying data change statement for a table containing an identity column. If a qualifying data change statement for a table containing an identity column was not executed within the same triggered action before invoking the IDENTITY_VAL_LOCAL function, then the function returns a null value.

Invoking the function following an insert operation with triggered actions: The result of invoking the function after an insert that activates triggers is the value actually assigned to the identity column (that is, the value that would be returned

on a subsequent SELECT statement). This value is not necessarily the value provided in the qualifying data change statement or a value generated by DB2. The assigned value could be a value that was specified in a SET transition variable statement within the triggered action of a before insert trigger for a trigger transition variable associated with the identity column.

Scope of IDENTITY_VAL_LOCAL: The IDENTITY_VAL_LOCAL value persists until the next insert in the current session into a table that has an identity column defined on it, or the application session ends. The value is unaffected by COMMIT or ROLLBACK statements for local applications. The IDENTITY_VAL_LOCAL value cannot be directly set and is a result of inserting a row into a table. Client applications or middleware products that save the state of a session and then restore the state of a session for subsequent processing are not able to restore the IDENTITY_VAL_LOCAL value. In these situations, the availability of the IDENTITY_VAL_LOCAL value should only be relied on until the end of the transaction. Examples of where this type of situation can occur include applications that do the following actions:

- use XA protocols
- use connection pooling
- use the connection concentrator
- use Sysplex workload balancing
- connect to a z/OS server that uses DDF inactive threads

When there is a need to preserve the value associated with IDENTITY_VAL_LOCAL across transaction boundaries for distributed applications, define the cursors as WITH HOLD, or specify the bind option KEEPDYNAMIC(YES) to prevent the server thread from being pooled.

Example 1: Set the variable *IVAR* to the value assigned to the identity column in the EMPLOYEE table. The value returned from the function in the VALUES statement should be 1.

```
CREATE TABLE EMPLOYEE
(EMPNO         INTEGER GENERATED ALWAYS AS IDENTITY,
 NAME          CHAR(30),
 SALARY        DECIMAL(5,2),
 DEPTNO        SMALLINT);
INSERT INTO EMPLOYEE
(NAME, SALARY, DEPTNO)
VALUES ('Rupert', 989.99, 50);
VALUES IDENTITY_VAL_LOCAL() INTO :IVAR;
```

Example 2: Assume two tables, T1 and T2, have an identity column named C1. DB2 generates values 1, 2, 3, . . . for the C1 column in table T1, and values 10, 11, 12, . . . for the C1 column in table T2.

```
CREATE TABLE T1 (C1 SMALLINT GENERATED ALWAYS AS IDENTITY,
                 C2 SMALLINT );
CREATE TABLE T2 (C1 DECIMAL(15,0) GENERATED BY DEFAULT AS IDENTITY
                                    (START WITH 10),
                 C2 SMALLINT );
INSERT INTO T1 (C2) VALUES (5);
INSERT INTO T1 (C2) VALUES (5);
SELECT * FROM T1;
    C1            C2
-----------   ----------

          1           5
          2           5
VALUES IDENTITY_VAL_LOCAL() INTO  :IVAR;
```

At this point, the IDENTITY_VAL_LOCAL function would return a value of 2 in IVAR. The following INSERT statement inserts a single row into T2 where column C2 gets a value of 2 from the IDENTITY_VAL_LOCAL function

```
INSERT INTO T2 (C2) VALUES (IDENTITY_VAL_LOCAL());
SELECT * FROM T2
 WHERE C1 = DECIMAL(IDENTITY_VAL_LOCAL(),15,0);
              C1                          C2
 --------------------------------   ----------

                                 10           2
```

Invoking the IDENTITY_VAL_LOCAL function after this insert would result in a value of 10, which is the value generated by DB2 for column C1 of T2. Assume another single row is inserted into T2. For the following INSERT statement, DB2 assigns a value of 13 to identity column C1 and gives C2 a value of 10 from IDENTITY_VAL_LOCAL. Thus, C2 is given the last identity value that was inserted into T2.

```
INSERT INTO T2 (C2, C1) VALUES (IDENTITY_VAL_LOCAL(), 13);
```

Example 3: The IDENTITY_VAL_LOCAL function can also be invoked in an INSERT statement that both invokes the IDENTITY_VAL_LOCAL function and causes a new value for an identity column to be assigned. The next value to be returned is thus established when the IDENTITY_VAL_LOCAL function is invoked after the INSERT statement completes. For example, consider the following table definition:

```
CREATE TABLE T1 (C1 SMALLINT GENERATED BY DEFAULT AS IDENTITY,
                 C2 SMALLINT);
```

For the following INSERT statement, specify a value of 25 for the C2 column, and DB2 generates a value of 1 for C1, the identity column. This establishes 1 as the value that will be returned on the next invocation of the IDENTITY_VAL_LOCAL function.

```
INSERT INTO T1 (C2) VALUES (25);
```

In the following INSERT statement, the IDENTITY_VAL_LOCAL function is invoked to provide a value for the C2 column. A value of 1 (the identity value assigned to the C1 column of the first row) is assigned to the C2 column, and DB2 generates a value of 2 for C1, the identity column. This establishes 2 as the value that will be returned on the next invocation of the IDENTITY_VAL_LOCAL function.

```
INSERT INTO T1 (C2) VALUES (IDENTITY_VAL_LOCAL());
```

In the following INSERT statement, the IDENTITY_VAL_LOCAL function is again invoked to provide a value for the C2 column, and the user provides a value of 11 for C1, the identity column. A value of 2 (the identity value assigned to the C1 column of the second row) is assigned to the C2 column. The assignment of 11 to C1 establishes 11 as the value that will be returned on the next invocation of the IDENTITY_VAL_LOCAL function.

```
INSERT INTO T1 (C2, C1) VALUES (IDENTITY_VAL_LOCAL(), 11);
```

After the 3 INSERT statements have been processed, table T1 contains the following actions:

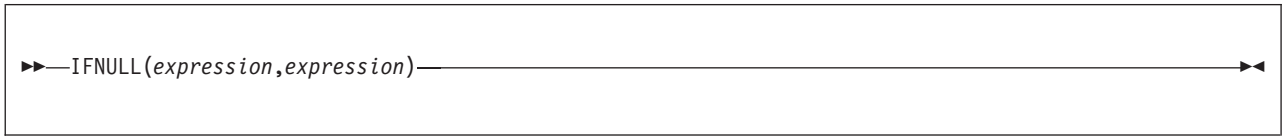
```
SELECT * FROM T1;
C1          C2
----------- -----------
```

```
  1          25
  2           1
 11           2
```

The contents of T1 illustrate that the expressions in the VALUES clause are evaluated before the assignments for the columns of the INSERT statement. Thus, an invocation of an IDENTITY_VAL_LOCAL function invoked from a VALUES clause of an INSERT statement uses the most recently assigned value for an identity column in a previous INSERT statement.

IFNULL

The IFNULL function returns the first nonnull expression.



The schema is SYSIBM.

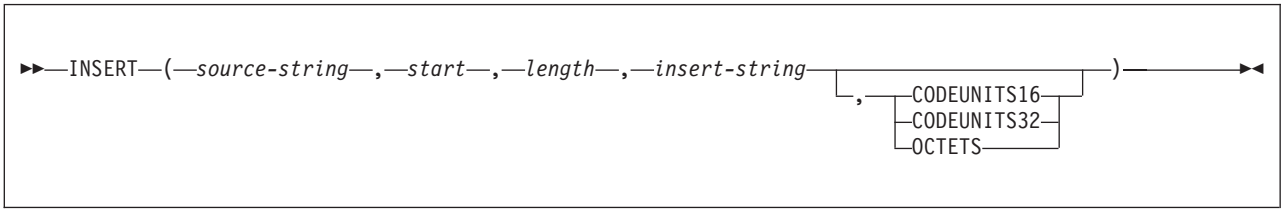
IFNULL is identical to the COALESCE scalar function except that IFNULL is limited to two arguments instead of multiple arguments. For a description, see "COALESCE" on page 412.

Example: For all the rows in sample table DSN8B10.EMP, select the employee number and salary. If the salary is missing (is null), have the value 0 returned.

```
SELECT EMPNO, IFNULL(SALARY,0)
  FROM DSN8B10.EMP;
```

INSERT

The INSERT function returns a string where, beginning at *start* in *source-string*, *length* characters have been deleted and *insert-string* has been inserted.



The schema is SYSIBM.

The INSERT function returns a string where *length* characters have been deleted from *source-string*, beginning at *start*, and where *insert-string* has been inserted into *source-string*, beginning at *start*.

source-string
An expression that specifies the source string. The expression must return a value that is a built-in character string, graphic string, or binary string data type that is not a LOB. The actual length of the string must be greater than or equal to 1 and less than or equal to 32704 bytes.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.

start
An expression that returns an integer. The integer specifies the starting point within the source string where the deletion of bytes and the insertion of another string is to begin. The value of the integer must be in the range of 1 to the length of *source-string* plus one. If OCTETS is specified and the result is graphic data, the value must be an odd value between 1 and twice the length of *source-string* plus one.

The argument can also be a character string or graphic string data type. The string argument is implicitly cast to a DECFLOAT(34) data type which is then assigned to an INTEGER.

length
An expression that specifies the length of the string to replace in *source-string* starting at *start*. *length* must be an expression that returns a value of the built-in INTEGER data type. *length* is expressed in the string unit specified, and the value must be in the range of 0 to the length of *source-string*. If OCTETS is specified and the result is graphic data, *length* must be even and be between 0 and twice the length of *source-string*. Not specifying *length* is equivalent to specifying a value of 1, except when OCTETS is specified and the result is graphic data, in which case, not specifying *length* is equivalent to specifying a value of 2.

The argument can also be a character string or graphic string data type. The string argument is implicitly cast to a DECFLOAT(34) data type which is then assigned to an INTEGER.

insert-string
An expression that specifies the string to be inserted into the source string,

starting at the position identified by *start*. The expression must return a value that is a built-in character string, graphic string, or binary string data type that is not a LOB.

source-string and *insert-string* must have compatible data types.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the units that are used to express *start* and *length*. If *source-string* is a character string that is defined as bit data, CODEUNITS16 and CODEUNITS32 cannot be specified. If *source-string* is a graphic string, OCTETS cannot be specified. If *source-string* is a binary string, CODEUNITS16, CODEUNITS32, and OCTETS cannot be specified.

If a string unit is not explicitly specified, the data type of the result determines the unit that is used:

- If the result is a graphic string, a string unit is two bytes. For ASCII and EBCDIC data, this corresponds to a double byte character. For Unicode, this corresponds to a UTF-16 code point.
- Otherwise, a string unit is a byte.

CODEUNITS16
Specifies that *start* and *length* are expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32
Specifies that *start* and *length* are expressed in terms of 32-bit UTF-32 code units.

OCTETS
Specifies that *start* and *length* are expressed in terms of bytes.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String unit specifications" on page 87.

If *source-string* and *insert-string* have different CCSID sets, *insert-string* (the string to be inserted) is converted to the CCSID of *source-string* (the source string).

The encoding scheme of the result is the same as *source-string*. The data type of the result of the function depends on the data type of *source-string* and *insert-string*:

- VARCHAR if *source-string* is a character string. The CCSID of the result depends on the arguments:
  - If either *source-string* or *insert-string* is character bit data, the result is bit data.
  - If *source-string* is SBCS Unicode data and *insert-string* is not SBCS Unicode data, the CCSID of the result is the mixed CCSID for Unicode data.
  - If both *source-string* and *insert-string* are SBCS Unicode data, the CCSID of the result is the CCSID for SBCS Unicode data.
  - Otherwise, the CCSID of the result is the mixed CCSID that corresponds to the CCSID of *source-string*. However, if the input is EBCDIC or ASCII and there is no corresponding system CCSID for mixed, the CCSID of the result is the CCSID of *source-string*.
- VARGRAPHIC if *source-string* is a graphic. The CCSID of the result is the same as the CCSID of *source-string*.
- VARBINARY if *source-string* and *insert-string* are both binary strings.

The length attribute of the result depends on the arguments:

- If *start* and *length* are constants, the length attribute of the result is:

  ```
  L1 - MIN((L1 - V2 + 1), V3) + L4
  ```

  where:
  L1 is the length attribute of *source-string*
  V2 is the value of *start*
  V3 is the value of *length*
  L4 is the length attribute of *insert-string*
- Otherwise, the length attribute of the result is the length attribute of *source-string* plus the length attribute of *insert-string*. In this case, the length attribute of *source-string* plus the length attribute of *insert-string* must not exceed 32704 for a VARCHAR result or 16352 for a VARGRAPHIC result.

If CODEUNITS16 or CODEUNITS32 is specified, the insert operation is performed on a Unicode version of the data. If needed, the data is converted to an intermediate form in order to evaluate the function. If an intermediate form is used, the actual length of the result depends on the original data (*source-string* and *insert-string*), and the representation of that data in Unicode. See "Determining the length attribute of the final result" on page 90 for more information on how to calculate the length attribute of the result string.

If CODEUNITS16 or CODEUNITS32 are not specified, the actual length of the result is:

```
A1 - MIN((A1 - V2 + 1), V3) + A4
```

where:
A1 is the actual length of *source-string*
V2 is the value of *start*
V3 is the value of *length*
A4 is the actual length of *insert-string*

If the actual length of the result string exceeds the maximum for the return data type, an error occurs.

The result can be null; if any argument is null, the result is the null value.

Example 1: The following example shows how the string 'INSERTING' can be changed into other strings. The use of the CHAR function limits the length of the resulting string to 10 bytes.

```
SELECT CHAR(INSERT('INSERTING',4,2,'IS'),10),
       CHAR(INSERT('INSERTING',4,0,'IS'),10),
       CHAR(INSERT('INSERTING',4,2,''),10)
  FROM SYSIBM.SYSDUMMY1;
```

This example returns 'INSISTING ', 'INSISERTIN', and 'INSTING '

Example 2: The previous example demonstrated how to insert text into the middle of some text. This example shows how to insert text before some text by using 1 as the starting point (*start*).

```
SELECT CHAR(INSERT('INSERTING',1,0,'XX'),10),
       CHAR(INSERT('INSERTING',1,1,'XX'),10),
       CHAR(INSERT('INSERTING',1,2,'XX'),10),
       CHAR(INSERT('INSERTING',1,3,'XX'),10)
  FROM SYSIBM.SYSDUMMY1;
```

This example returns 'XXINSERTIN', 'XXNSERTING', 'XXSERTING ', and 'XXERTING '

Example 3: The following example shows how to insert text after some text. Add 'XX' at the end of string 'ABCABC'. Because the source string is 6 characters long, set the starting position to 7 (one plus the length of the source string).

```
SELECT CHAR(INSERT('ABCABC',7,0,'XX'),10)
  FROM SYSIBM.SYSDUMMY1;
```

This example returns 'ABCABCXX '.

Example 4: The following example shows how the string 'Hegelstraße' can be changed to 'Hegelstrasse'.

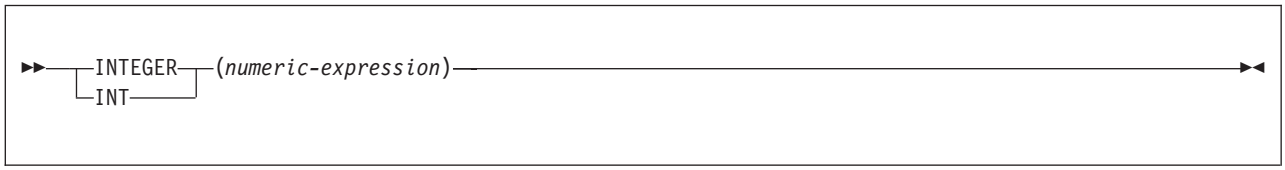
```
SELECT VARCHAR(INSERT('Hegelstraße',10,1,'ss'),15)
  FROM SYSIBM.SYSDUMMY1;
```

This example returns 'Hegelstrasse'.

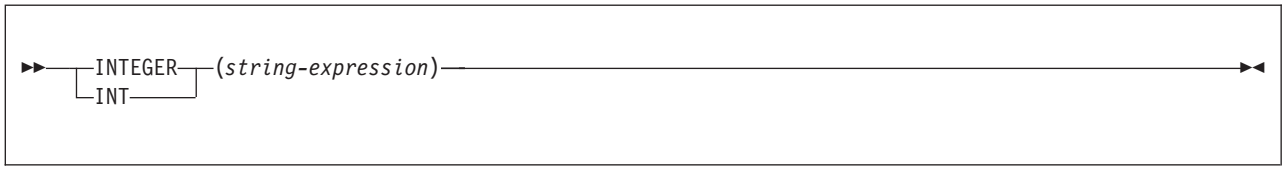
INTEGER or INT

The INTEGER function returns an integer representation of either a number or a character string or graphic string representation of an integer.

Numeric to Integer:



String to Integer:



The schema is SYSIBM.

Numeric to Integer

numeric-expression
An expression that returns a value of any built-in numeric data type.

The result is the same number that would occur if the argument were assigned to a large integer column or variable. If the whole part of the argument is not within the range of large integers, an error occurs. The fractional part of the argument is truncated.

String to Integer

string-expression
An expression that returns a value of a character or graphic string (except a CLOB or DBCLOB) with a length attribute that is not greater than 255 bytes. The string must contain a valid string representation of a number.

The result is the same number that would result from `CAST(`*string-expression*` AS INTEGER)`. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an integer constant. If the whole part of the argument is not within the range of large integers, an error is returned.

The result of the function is a large integer.

The result can be null; if the argument is null, the result is the null value.

Recommendation: To increase the portability of applications, use the CAST specification. For more information, see "CAST specification" on page 267.

Example 1: Using sample table DSN8B10.EMP, find the average salary of the employees in department A00, rounding the result to the nearest dollar.

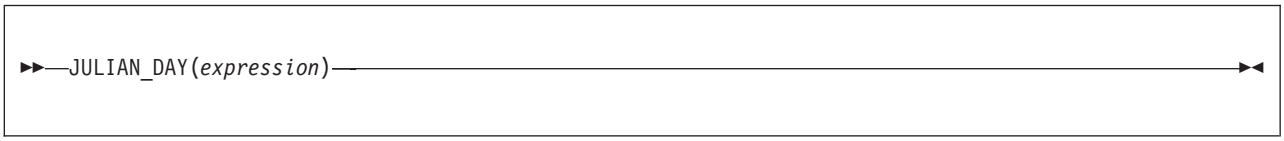
```
SELECT INTEGER(AVG(SALARY)+.5)
  FROM DSN8B10.EMP
  WHERE WORKDEPT = 'A00';
```

Example 2: Using sample table DSN8B10.EMP, select the EMPNO column, which is defined as CHAR(6), in integer form.

```
SELECT INTEGER(EMPNO)
  FROM DSN8B10.EMP;
```

JULIAN_DAY

The JULIAN_DAY function returns an integer value that represents a number of days from January 1, 4713 B.C. (the start of the Julian date calendar) to the date that is specified in the argument.



The schema is SYSIBM.

The argument must be an expression that returns one of the following data types: a date, a timestamp, or a valid string representation of a date or timestamp. An argument with a character string data type must not be a CLOB. An argument with a graphic string data type must not be a DBCLOB. A string argument must have an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 101.

If *expression* is a timestamp with a time zone, or a valid string representation of a timestamp with a time zone, the result is determined from the UTC representation of the datetime value.

The result of the function is a large integer.

The result can be null; if the argument is null, the result is the null value.

Example 1: Using sample table DSN8B10.EMP, set the integer host variable JDAY to the Julian day of the day that Christine Haas (EMPNO = '000010') was employed (HIREDATE = '1965-01-01').

```
SELECT JULIAN_DAY(HIREDATE)
  INTO :JDAY
  FROM DSN8B10.EMP
  WHERE EMPNO = '000010';
```

The result is that JDAY is set to 2438762.

Example 2: Set integer host variable JDAY to the Julian day for January 1, 1998.

```
SELECT JULIAN_DAY('1998-01-01')
  INTO :JDAY
  FROM SYSIBM.SYSDUMMY1;
```

The result is that JDAY is set to 2450815.

Example 3: The following invocations of the JULIAN_DAY function returns the same result:

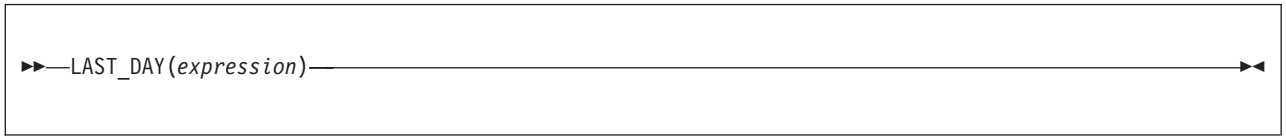
```
SELECT JULIAN_DAY('2003-01-02-20.00.00'),
   JULIAN_DAY('2003-01-02-12.00.00-08:00'),
   JULIAN_DAY('2003-01-03-05.00.00+09:00')
  FROM SYSIBM.SYSDUMMY1;
```

For each invocation of the JULIAN_DAY function in this SELECT statement, the result is 2452642.

When the input argument contains a time zone, the result is determined from the UTC representation of the input value. The string representations of a timestamp with a time zone in the SELECT statement all have the same UTC representation: 2003-01-02-20.00.00.

LAST_DAY

The LAST_DAY scalar function returns a date that represents the last day of the month of the date argument.



The schema is SYSIBM.

expression
An expression that specifies the starting date. The expression must returns a value of one of the following data types:

- a date
- a timestamp
- a valid string representation of a date or timestamp

An argument with a character string data type must not be a CLOB. An argument with a graphic string data type must not be a DBCLOB. A string argument must have an actual length that is not greater than 255 bytes. A time zone in a string representation of a timestamp is ignored. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 101.

If *expression* is a TIMESTAMP WITH TIME ZONE value, *expression* is first cast to a TIMESTAMP WITHOUT TIME ZONE value with the same precision as *expression*.

The result of the function is a DATE.

The result CCSID is the appropriate CCSID of the argument encoding scheme and the result subtype is the appropriate subtype of the CCSID.

The result can be null; if the argument is null, the result is the null value.

Any hours, minutes, seconds, or fractional seconds information that is included in *expression* is not changed by the function.

Example 1: Set the host variable *END_OF_MONTH* with the last day of the current month.

```
SET :END_OF_MONTH = LAST_DAY(CURRENT_DATE);
```

The host variable *END_OF_MONTH* is set with the value representing the end of the current month. If the current day is 2000-02-10, *END_OF_MONTH* is set to 2000-02-29.

Example 2: Set the host variable *END_OF_MONTH* with the last day of the month in EUR format for the given date.

```
SET :END_OF_MONTH = CHAR(LAST_DAY('1965-07-07'), EUR);
```

The host variable *END_OF_MONTH* is set with the value '31.07.1965'.

Example 3: Assume that host variable *PRSTSZ* contains '2008-02-29.20.00.000000 -08.30'. The TIMESTAMP WITH TIME ZONE value is implicitly cast to TIMESTAMP WITHOUT TIME ZONE before the LAST_DAY function is evaluated.

```
SELECT LAST_DAY(:PRSTSZ)
  FROM PROJECT;
```

The LAST_DAY function returns the value '31' (month in UTC is March).

Example 4: Assume *PRSTSZ* is a host variable with the string value '2008-04-15.20.00.000000-08.30'. The string value, which is a string representation of a timestamp with a time zone, is implicitly cast to a DATE before the LAST_DAY function is evaluated. The LAST_DAY function returns the last day of the month as a DATE value.

```
SELECT LAST_DAY(:PRSTSZ)
 FROM PROJECT;
```

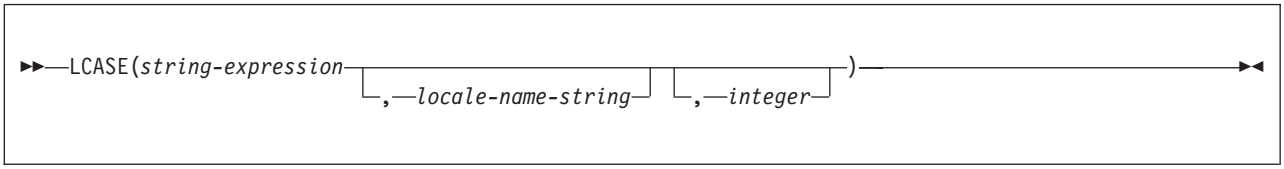
The LAST_DAY function returns the value ‘2008-04-30', the last day of the month of April, as a DATE value.

Example 5: Assuming that the default date format is ISO, the following select statement returns '2000–04–30', which is the last day of April in 2000:

```
SELECT LAST_DAY('2000-04-24')
 FROM SYSIBM.SYSDUMMY1;
```

LCASE

The LCASE function returns a string in which all the characters are converted to lowercase characters.



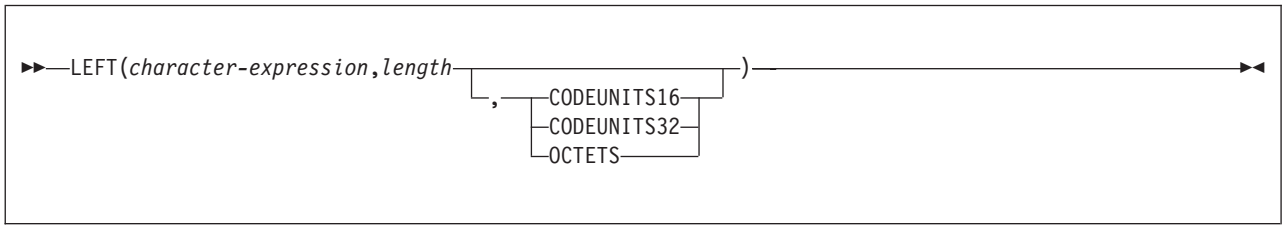
The schema is SYSIBM.

The LCASE function is identical to the LOWER function. For more information, see "LOWER" on page 517.

LEFT

The LEFT function returns a string that consists of the specified number of leftmost bytes of the specified string units.

Character string:



Graphic string:

Binary string:

The schema is SYSIBM.

The LEFT function returns the leftmost string of *character-expression*, *graphic-expression*, or *binary-expression* consisting of length of the string units that are specified implicitly or explicitly.

Character string:

character-expression
An expression that specifies the string from which the result is derived. The string must be a character string. A substring of *character-expression* is zero or more contiguous code points of *character-expression*.

The string can contain mixed data. Depending on the units that are specified to evaluate the function, the result is not necessarily a properly formed mixed data character string.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.

length
An expression that specifies the length of the result. The value must be an integer between 0 and *n*, where *n* is the length attribute of *character-expression*, expressed in the units that are either implicitly or explicitly specified.

The argument can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34) which is then assigned to an INTEGER value.

If CODEUNITS16 or CODEUNITS32 is specified, see "Determining the length attribute of the final result" on page 90 for information about how to calculate the length attribute of the result string.

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the unit that is used to express *length*. If *character-expression* is defined as bit data, CODEUNITS16 and CODEUNITS32 cannot be specified.

CODEUNITS16
Specifies that *length* is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32
Specifies that *length* is expressed in terms of 32-bit UTF-32 code units.

OCTETS
Specifies that *length* is expressed in terms of bytes.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String unit specifications" on page 87.

Graphic string:

graphic-expression
An expression that specifies the string from which the result is derived. The string must be a graphic string. A substring of *graphic-expression* is zero or more contiguous code points of *graphic-expression*. A partial surrogate character in the expression is replaced with a blank.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.

length
An expression that specifies the length of the result. The value must be an integer between 0 and *n*, where *n* is the length attribute of *graphic-expression*, expressed in the units that are either implicitly or explicitly specified.

The argument can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34) which is then assigned to an INTEGER value.

If CODEUNITS16 or CODEUNITS32 is specified, see "Determining the length attribute of the final result" on page 90 for information about how to calculate the length attribute of the result string.

CODEUNITS16 or CODEUNITS32
Specifies the unit that is used to express *length*.

CODEUNITS16
Specifies that *length* is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32
Specifies that *length* is expressed in terms of 32-bit UTF-32 code units.

For more information about CODEUNITS16 and CODEUNITS32, see "String unit specifications" on page 87.

Binary string:

binary-expression
An expression that specifies the string from which the result is derived. The string must be a binary string. A substring of *binary-expression* is zero or more contiguous code points of *binary-expression*.

length
An expression that specifies the length of the result. The value must be an integer between 0 and *n*, where *n* is the length attribute of *binary-expression*, expressed in the units that are either implicitly or explicitly specified.

The *character-expression*, *graphic-expression*, or *binary-expression* is effectively padded on the right with the necessary number of padding characters so that the specified substring of the expression always exists. The encoding scheme of the data determines the padding character:

- For ASCII SBCS data or ASCII mixed data, the padding character is X'20'.
- For ASCII DBCS data, the padding character depends on the CCSID; for example, for Japanese (CCSID 301) the padding character is X'8140', while for simplified Chinese it is X'A1A1'.
- For EBCDIC SBCS data or EBCDIC mixed data, the padding character is X'40'.
- For EBCDIC DBCS data, the padding character is X'4040'.
- For Unicode SBCS data or UTF-8 (Unicode mixed data), the padding character is X'20'.
- For UTF-16 (Unicode DBCS) data, the padding character is X'0020'.
- For binary data, the padding character is X'00'.

The result of the function is a varying-length string with a length attribute that is the same as the length attribute of the first expression and a data type that depends on the data type of the expression:

- VARCHAR if *character-expression* is CHAR or VARCHAR
- CLOB if *character-expression* is CLOB
- VARGRAPHIC if *graphic-expression* is GRAPHIC or VARGRAPHIC
- DBCLOB if *graphic-expression* is DBCLOB
- VARBINARY if *binary-expression* is BINARY or VARBINARY
- BLOB if *binary-expression* is BLOB

The actual length of the result is determined from *length*.

The result can be null; if any argument is null, the result is the null value.

The CCSID of the result is the same as that of the first expression.

Example 1: Assume that host variable *ALPHA* has a value of 'ABCDEF'. The following statement returns 'ABC', which are the three leftmost characters in *ALPHA*:

```
SELECT LEFT(:ALPHA,3)
  FROM SYSIBM.SYSDUMMY1;
```

Example 2: Assume that host variable *NAME*, which is defined as VARCHAR(50), has a value of 'KATIE AUSTIN' and the integer host variable FIRSTNAME_LEN has a value of 5. The following statement returns the value 'KATIE':

```
SELECT LEFT(:NAME, :FIRSTNAME_LEN)
  FROM SYSIBM.SYSDUMMY1;
```

Example 3: The following statement returns a zero length string.

```
SELECT LEFT('ABCABC',0)
  FROM SYSIBM.SYSDUMMY1;
```

Example 4: The FIRSTNME column in sample EMP table is defined as VARCHAR(12). Find the first name for an employee whose last name is 'BROWN' and return the first name in a 10-byte string.

```
SELECT LEFT(FIRSTNME,10)
  FROM DSN8B10.EMP
  WHERE LASTNAME='BROWN';
```

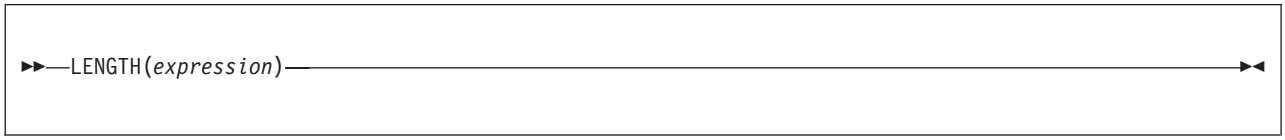
This function returns a VARCHAR(10) string that has the value of 'DAVID' followed by 5 blank characters.

Example 5: FIRSTNAME is a VARCHAR(12) column in table T1. One of its values is the 6-character string 'Jürgen'. When FIRSTNAME has this value:

```
Function ...                          Returns ...
LEFT(FIRSTNAME,2,CODEUNITS32)         'Jü'  -- x'4AC3BC'
LEFT(FIRSTNAME,2,CODEUNITS16)         'Jü'  -- x'4AC3BC'
LEFT(FIRSTNAME,2,OCTETS)              'J '  -- x'4A20' a truncated string
```

LENGTH

The LENGTH function returns the length of a value.



The schema is SYSIBM.

The argument must be an expression that returns a value of any built-in data type that is not XML.

The result of the function is a large integer.

The result can be null; if the argument is null, the result is the null value.

The result is the length of the argument. The length does not include the null indicator byte of column arguments that allow null values. The length of strings includes blanks. The length of a varying-length string is the actual length, not the maximum length.

The length of a graphic string is the number of double-byte characters. Unicode UTF-16 data is treated as graphic data; a UTF-16 supplementary character takes two DBCS characters to represent and as such is counted as two DBCS characters.

The length of all other values is the number of bytes used to represent the value:
- 2 for small integer
- 4 for large integer
- 8 for big integer
- The integer part of (p/2)+1 for decimal numbers with precision p
- 16 for DECFLOAT(34)
- 8 for DECFLOAT(16)
- 4 for single precision floating-point
- 8 for double precision floating-point
- The length of the string for strings
- 4 for DATE
- 3 for TIME
- 10 for TIMESTAMP
- 12 for TIMESTAMP WITH TIME ZONE
- 7+((p+1)/2) for TIMESTAMP(p)
- 9+((p+1)/2) for TIMESTAMP(p) WITH TIME ZONE
- The length of the row ID

Example 1: Assume that FIRSTNME is a VARCHAR(12) column that contains 'ETHEL' for employee 280. The following query returns the value 5:

```
SELECT LENGTH(FIRSTNME)
  FROM DSN8B10.EMP
  WHERE EMPNO = '000280';
```

Example 2: Assume that HIREDATE is a column of data type DATE. Then, regardless of value the following statement returns the value 4:

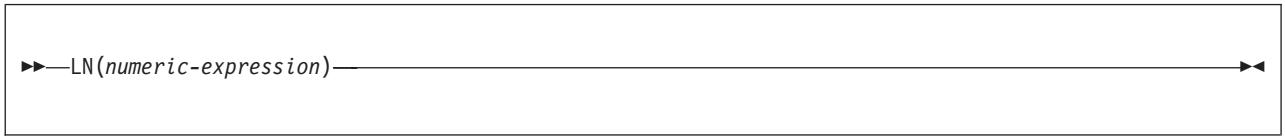
```
LENGTH(HIREDATE)
```

And the following function returns the value 10:

```
LENGTH(CHAR(HIREDATE, EUR))
```

LN

The LN function returns the natural logarithm of the argument. The LN and EXP functions are inverse operations.



The schema is SYSIBM.

The argument must be an expression that returns the value of any built-in numeric data type that is not DECFLOAT. If the argument is not a double precision floating-point number, it is converted to one for processing by the function.

The result of the function is a double precision floating-point number.

The result can be null; if the argument is null, the result is the null value.

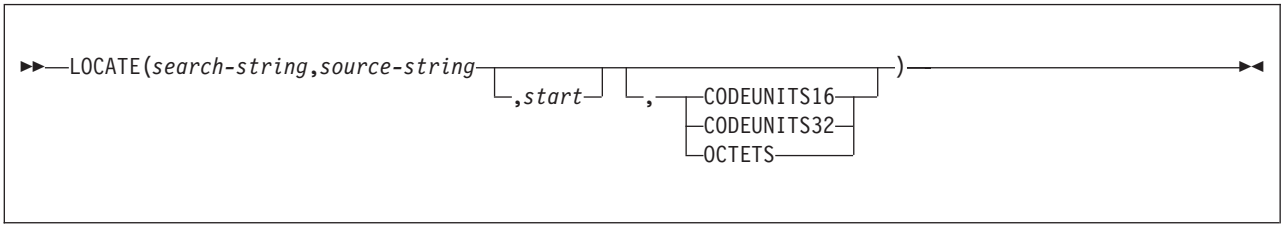
LOG is a synonym for LN.

Example: Assume that host variable NATLOG is DECIMAL(4,2) with a value of 31.62. The following statement returns a double precision floating-point number with an approximate value of 3.45:

```
SELECT LN(:NATLOG)
  FROM SYSIBM.SYSDUMMY1;
```

LOCATE

The LOCATE function returns the position at which the first occurrence of an argument starts within another argument.



The schema is SYSIBM.

The LOCATE function returns the starting position of *search-string* within *source-string*. If *search-string* is not found and neither argument is null, the result is zero. If *search-string* is found, the result is a number from 1 to the actual length of *source-string*. If *search-string* has a length of zero, the result returned by the function is 1. If the optional *start* is specified, it indicates the character position in *source-string* at which the search is to begin. An optional string unit can be specified to indicate in what units the start and result of the function are expressed.

search-string
An expression that specifies the string that is to be searched for. *search-string* must return a value that is a built-in character string data type, graphic string data type, or binary string data type with an actual length that is no greater than 4000 bytes.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.The expression can be specified by any of the following items:
- A constant
- A special register
- A variable
- A scalar function whose arguments are any of the above (though nested function invocations cannot be used)
- A CAST specification whose arguments are any of the above
- A column name
- An array element specification
- An expression that concatenates (using CONCAT or ||) any of the above

These rules are similar to those that are described for *pattern-expression* for the LIKE predicate.

source-string
An expression that specifies the source string in which the search is to take place. *source-string* must return a value that is a built-in character string data type, graphic string data type, or binary string data type.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.The expression can be specified by any of the following items:
- A constant
- A special register
- A variable

- A scalar function whose arguments are any of the above (though nested function invocations cannot be used)
- A CAST specification whose arguments are any of the above
- A column name
- An array element specification
- An expression that concatenates (using CONCAT or ||) any of the above

start
An expression that specifies the position within *search-string* where the search is to start.

start is expressed in the specified string unit and must return an integer value that is greater than or equal to zero.

The argument can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34) which is then assigned to an INTEGER value.

If *start* is specified, the LOCATE function is similar to the following POSITION function, where *string-units* is CODEUNITS16, CODEUNITS32, or OCTETS:

```
POSITION(search-string,
             SUBSTRING(source-string, start, string-units)) + start - 1
```

If *start* is not specified, the search begins at the first position of *source-string* and the LOCATE function is similar to the following POSITION function, where *string-units* is CODEUNITS16, CODEUNITS32, or OCTETS:

```
POSITION(search-string, source-string, string-units)
```

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the string unit that is used to express *start* and the result. If *source-string* is a character string that is defined as bit data, CODEUNITS16 and CODEUNITS32 cannot be specified. If *source-string* is a graphic string, OCTETS cannot be specified. If *source-string* is a binary string, CODEUNITS16, CODEUNITS32, and OCTETS cannot be specified.

CODEUNITS16
Specifies that *start* and the result are expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32
Specifies that *start* and the result are expressed in terms of 32-bit UTF-32 code units.

OCTETS
Specifies that *start* and the result are expressed in terms of bytes.

If a string unit is not explicitly specified, the data type of the result determines the string unit that is used. If the result is graphic data, *start* and the returned position are expressed in two-byte units; otherwise, they are expressed in bytes.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String unit specifications" on page 87.

The first and second arguments must have compatible string types. For more information on compatibility, see "Conversion rules for comparisons" on page 138.

The result of the function is a large integer.

The result can be null; if any argument is null, the result is the null value.

For more information about LOCATE, see the description of “POSITION” on page 566.

Example 1: Find the location of the first occurrence of the character 'N' in the string 'DINING'.

```
SELECT LOCATE('N', 'DINING')
  FROM SYSIBM.SYSDUMMY1;
```

The result is the value 3.

Example 2: For all the rows in the table named IN_TRAY, select the RECEIVED column, the SUBJECT column, and the starting position of the string 'GOOD' within the NOTE_TEXT column.

```
SELECT RECEIVED, SUBJECT, LOCATE('GOOD', NOTE_TEXT)
  FROM IN_TRAY
  WHERE LOCATE('GOOD', NOTE_TEXT) <> 0;
```

Example 3: Locate the character 'ß' in the string 'Jürgen lives on Hegelstraße', and set the host variable LOCATION with the position, as measured in CODEUNITS32 units, within the string.

```
SET :LOCATION = LOCATE('ß','Jürgen lives on Hegelstraße',1,CODEUNITS32);
```

The value of host variable LOCATION is set to 26.

Example 4: Locate the character 'ß' in the string 'Jürgen lives on Hegelstraße', and set the host variable LOCATION with the position, as measured in CODEUNITS16 units, within the string.

```
SET :LOCATION = LOCATE('ß','Jürgen lives on Hegelstraße',1,CODEUNITS16);
```

The value of host variable LOCATION is set to 26.

Example 5: Locate the character 'ß' in the string 'Jürgen lives on Hegelstraße', and set the host variable LOCATION with the position, as measured in OCTETS, within the string.

```
SET :LOCATION = LOCATE('ß','Jürgen lives on Hegelstraße',1,OCTETS);
```

The value of host variable LOCATION is set to 27.

Related reference:

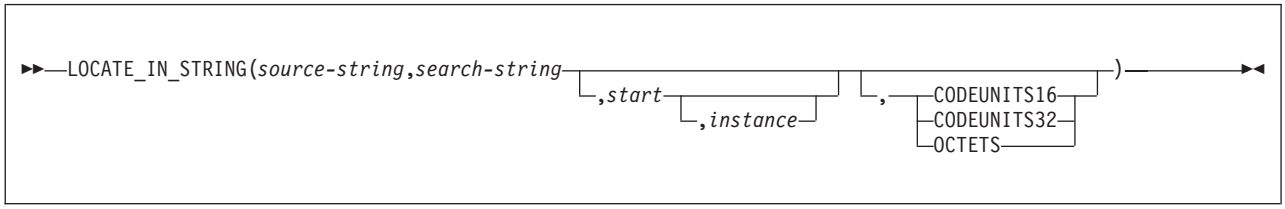
“LOCATE_IN_STRING” on page 513

“POSITION” on page 566

“POSSTR” on page 569

LOCATE_IN_STRING

The LOCATE_IN_STRING function returns the position at which an argument starts within a specified string.



The schema is SYSIBM.

The LOCATE_IN_STRING function returns the starting position of a string (called the *search-string*) within another string (called the *source-string*). If the *search-string* is not found and neither argument is null, the result is zero. If the *search-string* is found, the result is a number from 1 to the actual length of the *source-string*.

If the optional *start* is specified, an optional instance number can also be specified. The instance argument is used to determine the specific occurrence of *search-string* within *source-string*. Each unique instance can include any of the characters in a previous instance, but not all characters in a previous instance. An optional string unit can be specified to indicate in what units the *start* and result of the function are expressed.

If the *search-string* has a length of zero, the result returned by the function is 1. If the *source-string* has a length of zero, the result returned by the function is 0. If neither condition exists, and if the value of *search-string* is equal to an identical length of a substring of contiguous positions within the value of *source-string*, the result returned by the function is the starting position of that substring within the *source-string* value; otherwise, the result returned by the function is 0.

source-string
An expression that specifies the source string in which the search is to take place. *source-string* must return a value that is a built-in character string data type, graphic string data type, or binary string data type.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.The expression can be specified by any of the following items:

- A constant
- A special register
- A variable
- A scalar function whose arguments are any of the above (though nested function invocations cannot be used)
- A CAST specification whose arguments are any of the above
- A column name
- An array element specification
- An expression that concatenates (using CONCAT or ||) any of the above

search-string
An expression that specifies the string that is the object of the search. *search-string* must return a value that is a built-in character string data type, graphic string data type, or binary string data type with an actual length that is no greater than 4000 bytes.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.The expression can be specified by any of the following items:

- A constant
- A special register
- A variable
- A scalar function whose arguments are any of the above (though nested function invocations cannot be used)
- A CAST specification whose arguments are any of the above
- A column name
- An array element specification
- An expression that concatenates (using CONCAT or ||) any of the above

These rules are similar to those that are described for *pattern-expression* for the LIKE predicate.

start
An expression that specifies the position within *source-string* at which the search is to start. The expression must return a value that is a built-in INTEGER or SMALLINT data type.

The argument can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34) which is then assigned to an INTEGER value.

If the value of the integer is greater than zero, the search begins at *start* and continues for each position to the end of the string. If the value of the integer is less than zero, the search begins at the LENGTH(*source-string*) + *start* + 1 and continues for each position to the beginning of the string.

If *start* is not specified, the default is 1. If the value of the integer is zero, an error is returned.

instance
An expression that specifies which instance of *search-string* to search for within *source-string*. The expression must return a value that is a built-in INTEGER or SMALLINT data type. If *instance* is not specified, the default is 1. The value of the integer must be greater than or equal to one.

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the string unit that is used to express *start* and the result. If *source-string* is a character string that is defined as bit data, CODEUNITS16 and CODEUNITS32 cannot be specified. If *source-string* is a graphic string, OCTETS cannot be specified. If *source-string* is a binary string, CODEUNITS16, CODEUNITS32, and OCTETS cannot be specified.

CODEUNITS16
Specifies that *start* and the result are expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32
Specifies that *start* and the result are expressed in terms of 32-bit UTF-32 code units.

OCTETS
Specifies that *start* and the result are expressed in terms of bytes.

If a string unit is not explicitly specified, the data type of the result determines the string unit that is used. If the result is graphic data, *start* and the returned position are expressed in two-byte units; otherwise, they are expressed in bytes.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String unit specifications" on page 87.

The first and second arguments must have compatible string types. For more information on compatibility, see "Conversion rules for comparisons" on page 138.

At each search position, a match is found when the substring at that position and LENGTH(*search-string*) - 1 values to the right of the search position in *source-string*, is equal to *search-string*.

The result of the function is a large integer. The result is the starting position of the instance of *search-string* within *source-string*. The value is relative to the beginning of the string (regardless of the specification of *start*).

The result can be null; if any argument is null, the result is the null value.

INSTR can be used as a synonym for LOCATE_IN_STRING.

Example 1: Find the position of an occurrence of the character 'N' in the string 'WINNING' by searching from the start of the string as measured in bytes, within the string.

```
SELECT LOCATE_IN_STRING('WINNING','N',1,3,OCTETS),
  LOCATE_IN_STRING('WINNING','N',3,2,OCTETS),
  LOCATE_IN_STRING('WINNING','N',3,3,OCTETS),
  LOCATE_IN_STRING('WINNING','N',-1,3,OCTETS),
  LOCATE_IN_STRING('WINNING','N',-3,2,OCTETS),
  LOCATE_IN_STRING('WINNING','N',-3,3,OCTETS)
FROM SYSIBM.SYSDUMMY1;
```

Returns the values:

```
6 4 6 3 3 0
```

Related reference:

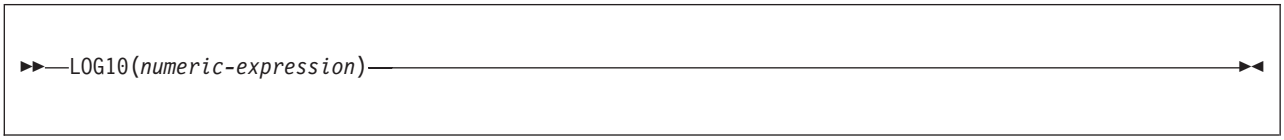
"LOCATE" on page 510

"POSITION" on page 566

"POSSTR" on page 569

LOG10

The LOG10 function returns the common logarithm (base 10) of a number.



The schema is SYSIBM.

The argument is an expression that returns the value of any built-in numeric data type that is not DECFLOAT. If the argument is not a double precision floating-point number, it is converted to one for processing by the function.

The result of the function is a double precision floating-point number.

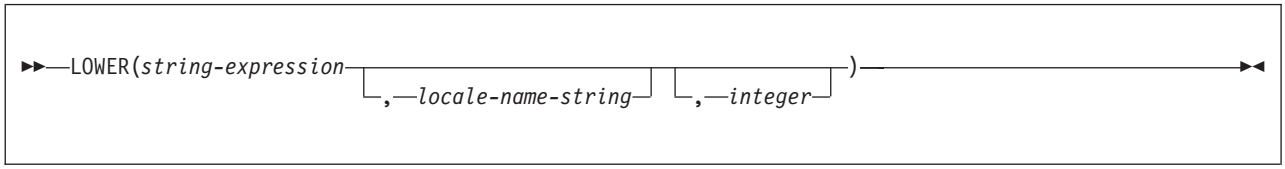
The result can be null; if the argument is null, the result is the null value.

Example: Assume that host variable HLOG is an INTEGER with a value of 100. The following statement returns a double precision floating-point number with an approximate value of 2:

```
SELECT LOG10(:HLOG)
  FROM SYSIBM.SYSDUMMY1;
```

LOWER

The LOWER function returns a string in which all the characters are converted to lowercase characters.



The schema is SYSIBM.

string-expression
An expression that specifies the string to be converted. *string-expression* must return a value that is a built-in character or graphic string. A character string argument must not be a CLOB, and a graphic string argument must not be a DBCLOB. If *string-expression* is an EBCDIC graphic string, a blank string must not be specified for *locale-name-string*.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.

locale-name-string
A string constant or a string host variable other than a CLOB or DBCLOB that specifies a valid locale name. If *locale-name-string* is not in EBCDIC, it is converted to EBCDIC. The length of *locale-name-string* must be between 1 and 255 bytes of the EBCDIC representation. The value of *locale-name-string* is not case sensitive and must be a valid locale. For information on locales and their naming conventions, see *z/OS C/C++ Programming Guide*. Some examples of locales include:
Fr_BE
Fr_FR@EURO
En_US
Ja_JP

The conversion process is determined by the value that is specified for the locale name, as follows:

- blank — SBCS uppercase characters A-Z are converted to SBCS lowercase characters a-z, and characters with diacritical marks are not converted. If the string contains MIXED or DBCS characters, full-width Latin uppercase characters A-Z are converted to full-width lowercase characters a-z. For optimal performance, specify a blank string unless your data must be processed by using the rules that are defined by a specific locale.
- UNI — The conversion uses both the NORMAL and SPECIAL casing capabilities as described in *z/OS Support for Unicode: Using Conversion Services*. You must not specify UNI when *string-expression* is EBCDIC data.
- locale name — The locale defines the rules for conversion to lowercase characters.

The value of the host variable must not be null. If the host variable has an associated indicator variable, the value of the indicator variable must not indicate a null value. The locale name must be:

- left justified within the host variable

- padded on the right with blanks if its length is less than that of the host variable and the host variable is in fixed length character or graphic data type

If *locale-name-string* is not specified, the locale is determined by special register CURRENT LOCALE LC_CTYPE. For information about the special register, see "CURRENT LOCALE LC_CTYPE" on page 177. However, if an index references the LOWER function, the local is determined as follows (in order) to determine if the index can be used:

- At prepare time — using the value in the CURRENT LOCALE LC_CTYPE special register
- At bind time — using the value in the LOCALE LC_CTYPE field on installation panel DSNTIPF

If the index is chosen in the access path, the locale in the CURRENT LOCALE LC_CTYPE special register must remain the same at run time, and prepare or bind time. To avoid this dependency, do not omit *locale-name-string*.

If the LOWER function is referenced in an expression-based index, *locale-name-string* must be specified. See the examples section for an example of how the index can be used in a query.

integer
An integer value that specifies the length attribute of the result. If specified, *integer* must be an integer constant between 1 and 32704 bytes in the representation of the encoding scheme of *string-expression*.

If *integer* is not specified, the length attribute of the result is the same as the length of *string-expression*.

For Unicode data, usage of the LOWER function can result in expansion if certain characters are processed. For example, LOWER ('Ì') —UX'00CC'— will result in UX'006903070300' (if the LT_LT locale is in effect at the time). You should ensure that the result length is large enough to contain the result of the expression.

The result can be null; if the argument is null, the result is the null value.

LCASE is a synonym for LOWER.

Example 1: Return the characters in the value of host variable NAME in lowercase. NAME has a data type of VARCHAR(30) and a value of 'Christine Smith'. Assume that the locale in effect is blank.

```
SELECT LCASE(:NAME)
  FROM SYSIBM.SYSDUMMY1;
```

The result is the value 'christine smith'.

Example 2: Return the lowercase of 'Ì'. Assume that the locale in effect is LT_LT.

```
SELECT LOWER('Ì')
  FROM SYSIBM.SYSDUMMYU;
```

This would result in an error because of the expansion that occurs when certain Unicode characters are processed. To avoid the error, you would need to use the following statement instead:

```
SELECT LOWER(VARCHAR('Ì', 3))
  FROM SYSIBM.SYSDUMMYU;
```

The result of the preceding statement is the value UX'006903070300'.

Example 3: Create an index EMPLOYEE_NAME_LOWER for table EMPLOYEE based on built-in function LOWER with locale name 'LT_LT'.

```
CREATE INDEX EMPLOYEE_NAME_LOWER
  ON EMPLOYEE (LOWER(LASTNAME, 'LT_LT', 60),
               LOWER(FIRSTNAME, 'LT_LT', 60),
               ID);
```

Example 4: Create an index LNAME for table T1 based on the LOWER function with the default local value, ' '. Then specify the same expression in a query.

```
CREATE INDEX LNAME
  ON TI (LOWER(LASTNAME, ' '));

SELECT LOWER(LASTNAME, ' ')
  FROM TI
  WHERE LOWER(LASTNAME, ' ') = 'smith';
```

Example 5: Create an index LNAME that is based on the LOWER function with a locale name 'FR_CA' for the table T1. Then specify the same expression in a query except *locale-name-string* is omitted.

```
CREATE INDEX LNAME
  ON TI (LOWER(LASTNAME, 'FR_CA'));
```

If the query is a dynamic statement and the CURRENT LOCALE LC_CTYPE special register contains 'FR_CA':

```
SELECT LASTNAME
  FROM TI
  WHERE LOWER(LASTNAME)='smith';
```

At prepare time, locale 'FR_CA' in CURRENT LOCALE LC_CTYPE is used for LOWER(LASTNAME) in the predicate to determine whether index LNAME can be used for index access. If index LNAME is used in access path selection, at run time, the locale in CURRENT LOCALE LC_CTYPE must remain the same.

If the query is a static statement and locale 'FR_CA' has been set on the LOCALE LC_CTYPE field of installation panel DSNTIPF:

```
SELECT LASTNAME
  FROM TI
  WHERE LOWER(LASTNAME)='smith';
```

At bind time, local 'FR_CA' in the LOCALE LC_CTYPE file of installation panel DSNTIPF is used for LOWER(LASTNAME) in the predicate to determine whether index LNAME is used for index access. If index LNAME is chosen in access path selection, the locale in the CURRENT LOCALE LC_CTYPE special register must contain 'FR_CA'.

Related concepts:

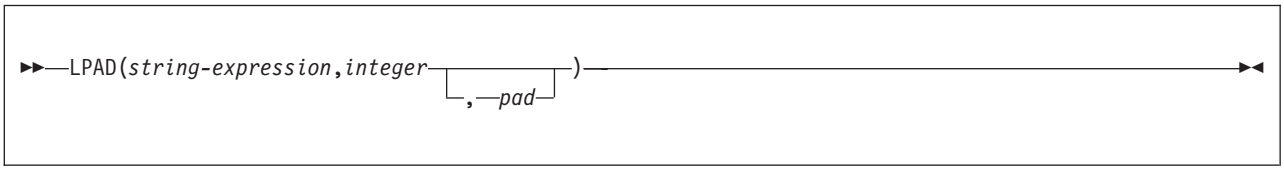
z/OS: Unicode Services User's Guide and Reference

Related reference:

z/OS XL C/C++ Programming Guide

LPAD

The LPAD function returns a string that is composed of *string-expression* that is padded on the left, with *pad* or blanks. The LPAD function treats leading or trailing blanks in *string-expression* as significant.



Padding occurs only if the actual length of *string-expression* is less than *integer*, and if *pad* is not an empty string.

The schema is SYSIBM.

string-expression
An expression that specifies the source string. The expression must return a value that is a built-in string data type that is not a LOB.

integer
An integer constant that specifies the length of the result. The value must be zero or a positive integer that is less than or equal to *n*, where *n* is 32704 if *string-expression* is a character or binary string, or where *n* is 16352 if *string-expression* is a graphic string.

pad
An expression that specifies the string with which to pad. The expression must return a value that is a built-in string data type that is not a LOB. If *pad* is not specified, the pad character is determined as follows:

- SBCS blank character if *string-expression* is a character string.
- DBCS blank character if *string-expression* is a graphic string.
- Hexadecimal zero (X'00'), if *string-expression* is a binary string.

The result of the function is a varying length string that has the same CCSID of *string-expression*. *string-expression* and *pad* must have compatible data types. If the string expressions have different CCSID sets, then *pad* is converted to the CCSID set of *string-expression*. If either *string-expression* or *pad* is FOR BIT DATA, no character conversion occurs.

The length attribute of the result depends on *integer*. If *integer* is greater than 0, the length attribute of the result is *integer*. If *integer* is 0, the length attribute of the result is 1.

The actual length of the result is determined from *integer*. If *integer* is 0, the actual length is 0, and the result is the empty result string. If *integer* is less than the actual length of *string-expression*, the actual length is *integer* and the result is truncated.

The result can be null; if any argument is null, the result is the null value.

Example 1: Assume that NAME is a VARCHAR(15) column that contains the values 'Chris', 'Meg', and 'Jeff'. The following query will pad a value on the left with periods.

```
SELECT LPAD(NAME,15,'.' ) AS NAME
  FROM T1;
```

The results are similar to the following output:

```
NAME
---------------

.........Chris
...........Meg
..........Jeff
```

Example 2: Similar to Example 1, the following query will only pad each value to a length of 5:

```
SELECT LPAD(NAME,5,'.' ) AS NAME
  FROM T1;
```

The results are similar to the following output:

```
NAME
---------------

Chris
..Meg
.Jeff
```

Example 3: Assume that NAME is a CHAR(15) column containing the values 'Chris', 'Meg', and 'Jeff. 'Note that the LPAD function does not pad because NAME is a fixed length character field and is blank padded already. However, since the length of the result is 5, the columns are truncated:

```
SELECT LPAD(NAME,5,'.' ) AS NAME
  FROM T1;
```

The results are similar to the following output:

```
NAME
---------------

Chris
Meg
Jeff
```

Example 4: Assume that NAME is a VARCHAR(15) column containing the values 'Chris', 'Meg', and 'Jeff'. Note that in some cases, a partial instance of the pad specification is returned.

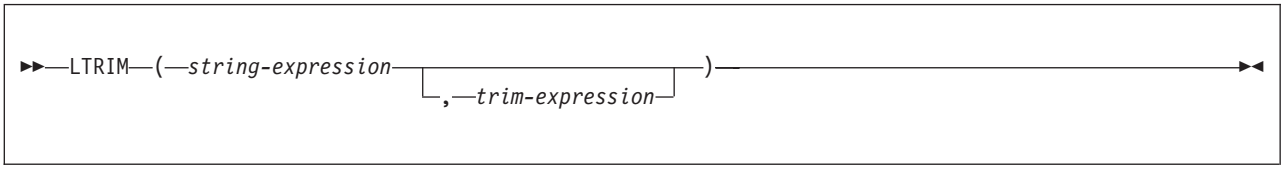
```
SELECT LPAD(NAME,15,'123') AS NAME
  FROM T1
```

The results are similar to the following output:

```
NAME
---------------
1231231231Chris
123123123123Meg
12312312312Jeff
```

LTRIM

The LTRIM function removes bytes from the beginning of a string expression based on the content of a trim expression.



The schema is SYSIBM.

The LTRIM function removes all of the characters that are contained in *trim-expression* from the beginning of *string-expression*. The search is done by comparing the binary representation of each character (which consists of one or more bytes) in *trim-expression* to the bytes at the beginning of *string-expression*. If the *string-expression* is defined as FOR BIT DATA, the search is done by comparing each byte in *trim-expression* to the byte at the beginning of *string-expression*.

string-expression
An expression that specifies the source string. The argument must be an expression that returns a value that is a built-in string data type that is not a LOB, or a numeric data type. If the value is not a string data type, it is implicitly cast to VARCHAR before the function is evaluated. If *string-expression* is not FOR BIT DATA, *trim-expression* must not be FOR BIT DATA.

trim-expression
An expression that specifies the characters to remove from the beginning of *string-expression*. The expression must return a value that is a built-in string data type that is not a LOB, or a numeric data type. If the value is not a string data type, it is implicitly cast to VARCHAR before the function is evaluated.

The default for *trim-expression* depends on the data type of *string-expression*:

- A DBCS blank if *string-expression* is a DBCS graphic string. For ASCII, the CCSID determines the hex value that represents a DBCS blank. For example, for Japanese (CCSID 301), X'8140' represents a DBCS blank, while for Simplified Chinese, X'A1A1' represents a DBCS blank. For EBCDIC, X'4040' represents a DBCS blank.
- A UTF-16 or UCS-2 blank (X'0020') if *string-expression* is a Unicode graphic string.
- A value of X'00' if *string-expression* is a binary string.
- Otherwise, a single byte blank. For EBCDIC, X'40' represents a blank. If not EBCDIC, X'20' represents a blank.

string-expression and *trim-expression* must have compatible data types. If *string-expression* and *trim-expression* have different CCSID sets, *trim-expression* is converted to the CCSID of *string-expression*.

The result of the function depends on the data type of *string-expression*:

- VARCHAR if *string-expression* is a character string. If *string-expression* is defined as FOR BIT DATA the result is FOR BIT DATA.
- VARGRAPHIC if *string-expression* is a graphic string.

- VARBINARY if *string-expression* is a binary string.

The length attribute of the result is the same as the length attribute of *string-expression*.

The actual length of the result for a character or binary string is the length of *string-expression* minus the number of bytes that are removed. The actual length of the result for a graphic string is the length (in the number of double byte characters) of *string-expression* minus the number of double byte characters removed. If all of the characters bytes are removed, the result is an empty string (the length is zero).

The result can be null; if the argument is null, the result is the null value.

The CCSID of the result is the same as that of *string-expression*.

Example: **Use the LTRIM function to remove individual numbers in the second argument from the beginning (left side) of the first argument:**

```
SELECT LTRIM ('123DEFG123', '321'),
    LTRIM ('12DEFG123', '321'),
    LTRIM ('123123222XYZ22', '123'),
    LTRIM ('12321', '213'),
    LTRIM ('XYX123 ', '321')
   FROM SYSIBM.SYSDUMMY1
```

The result is 'DEFG123', 'DEFG123', 'XYZ22', '' (an empty string - all characters removed), and 'XYX123' (no characters removed).

The LTRIM function does not remove instances of '1', '2', and '3' on the right side of the string, following characters that are not '1', '2', or '3'.

Example: **Use the LTRIM function to remove individual special characters in the second argument from the beginning (left side) of the first argument:**

```
SELECT LTRIM ('[[ -78]]', '- []')
   FROM SYSIBM.SYSDUMMY1
```

The result is '78]]'.

Example: **Use the LTRIM function to remove dollar signs and periods in the second argument from the beginning (left side) of the first argument:**

```
SELECT LTRIM ('...$V..$AR', '$.')
   FROM SYSIBM.SYSDUMMY1
```

The result is 'V..$AR'.

Example: **Use the LTRIM function to trim full multi-byte X'D090' characters:**
Assume that these strings are encoded in UTF-8.

```
SELECT LTRIM (X'D090D091D092', X'D090')
   FROM SYSIBM.SYSDUMMY1
```

The result is X'D091D092'.

Note that the function does not remove individual bytes x'D0' and x'90'.

MAX

The MAX scalar function returns the maximum value in a set of values.

The schema is SYSIBM.

The arguments must be compatible. For more information on compatibility, refer to the compatibility matrix in Table 23 on page 121. All but the first argument can be parameter markers. There must be two or more arguments.

Each argument must be an expression that returns a value of any built-in data type other than a CLOB, DBCLOB, BLOB, ROWID, or XML.

Character string arguments and binary string arguments cannot have a length attribute greater than 32704, and graphic string arguments cannot have a length attribute greater than 16352.

The arguments are evaluated in the order in which they are specified. The result of the function is the maximum argument value.

The result can be null; if any argument is null, the result is the null value.

The selected argument is converted, if necessary, to the attributes of the result. The attributes of the result are determined using the "Rules for result data types" on page 144. If the MAX function has more than two arguments, the rules are applied to the first two arguments to determine a candidate result type. The rules are then applied to that candidate result type and the third argument to determine another candidate result type. This process continues until all arguments are analyzed and the final result type and CCSID is determined.

GREATEST can be specified as a synonym for MAX.

Example 1: Assume the host variable *M1* is a DECIMAL(2,1) host variable with a value of 5.5, host variable *M2* is a DECIMAL(3,1) host variable with a value of 4.5, and host variable *M3* is a DECIMAL(3,2) host variable with a value of 6.25. The following function returns the value 6.25.

```
MAX(:M1,:M2,:M3)
```

Example 2: Assume the host variable *M1* is a CHAR(2) host variable with a value of 'AA', host variable *M2* is a CHAR(3) host variable with a value of 'AA ', and host variable *M3* is a CHAR(4) host variable with a value of 'AA A'. The following function returns the value 'AA A'.

```
MAX(:M1,:M2,:M3)
```

MAX_CARDINALITY

The MAX_CARDINALITY function returns a value of type BIGINT that represents the maximum number of elements that an array can contain. This value is the cardinality that was specified in the CREATE TYPE statement for an ordinary array type.

The schema is SYSIBM.

array-expression
An SQL variable or SQL parameter of an array type, or a CAST specification of a parameter marker to an array type.

The result of the MAX_CARDINALITY function is as follows:

- For an ordinary array, the result is the maximum number of elements that an array can contain.
- For an associative array, the result is the null value.

The data type of the result is BIGINT.

The result can be null; if the argument is null, the result is the null value.

Notes

Syntax alternatives: CAST (*SQL-variable* AS *array-type*) can be specified as an alternative to *SQL-variable*. CAST (*SQL-parameter* AS *array-type*) can be specified as an alternative to *SQL-parameter*.

Example 1: Suppose that array type PHONENUMBERS and array variable RECENT_CALLS are defined as follows:

```
CREATE TYPE PHONENUMBERS AS DECIMAL(10,0) ARRAY[50];
DECLARE RECENT_CALLS PHONENUMBERS;
```

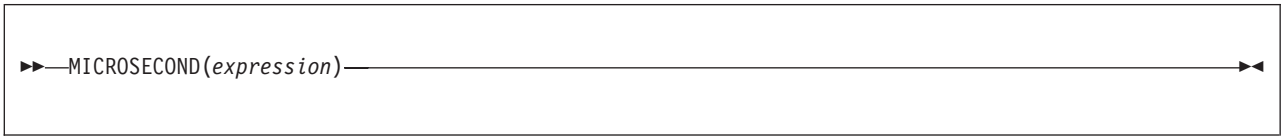
The following statement sets LIST_SIZE to the maximum cardinality with which RECENT_CALLS was defined.

```
SET LIST_SIZE = MAX_CARDINALITY(RECENT_CALLS);
```

After the statement executes, LIST_SIZE contains 50.

MICROSECOND

The MICROSECOND function returns the microsecond part of a value.



The schema is SYSIBM.

The argument must be an expression that returns a value of one of the following built-in data types: a timestamp, a character string, a graphic string, or a numeric data type.

- If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a timestamp with an actual length of not greater than 255 bytes. For the valid formats of string representations of times and timestamps, see "String representations of datetime values" on page 101.
- If *expression* is a number, it must be a timestamp duration. For the valid formats of timestamp durations, see "Datetime operands" on page 147.

If *expression* is a timestamp with a time zone, or a valid string representation of a timestamp with a time zone, the result is determined from the UTC representation of the datetime value.

The result of the function is a large integer.

The result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

If the argument is a timestamp or string representation of a timestamp, the result is the microsecond part of the value, which is an integer between 0 and 999999. If the precision of the timestamp exceeds 6, the value is truncated.

If the argument is a duration, the result is the microsecond part of the value, which is an integer between -999999 and 999999. A nonzero result has the same sign as the argument.

Example 1: Assume that table TABLEX contains a TIMESTAMP column named TSTMPCOL and a SMALLINT column named INTCOL. Select the microseconds part of the TSTMPCOL column of the rows where the INTCOL value is 1234:

```
SELECT MICROSECOND(TSTMPCOL) FROM TABLEX
  WHERE INTCOL = 1234;
```

Example 2: The following invocations of the MICROSECOND function returns the same result:

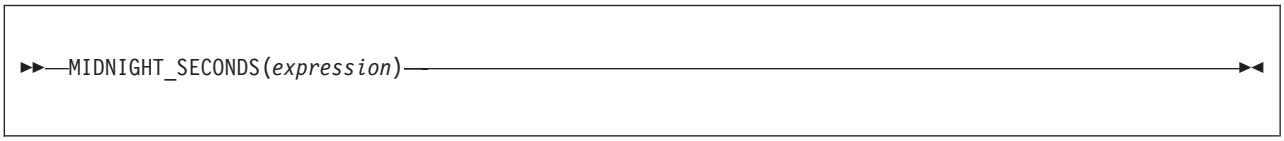
```
SELECT MICROSECOND('2003-01-02-20.00.00.123456'),
   MICROSECOND('2003-01-02-12.00.00.123456-08:00'),
   MICROSECOND('2003-01-03-05.00.00.123456+09:00')
  FROM SYSIBM.SYSDUMMY1;
```

For each invocation of the MICROSECOND function in this SELECT statement, the result is 123456.

When the input argument contains a time zone, the result is determined from the UTC representation of the input value. The string representations of a timestamp with a time zone in the SELECT statement all have the same UTC representation: 2003-01-02-20.00.00.123456.

MIDNIGHT_SECONDS

The MIDNIGHT_SECONDS function returns an integer, in the range of 0 to 86400, that represents the number of seconds between midnight and the time that is specified in the argument.



The schema is SYSIBM.

The argument must be an expression that returns a value of one of the following built-in data types: a time, a timestamp, a character string, or a graphic string. If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a time or timestamp with an actual length of not greater than 255 bytes. For the valid formats of string representations of times and timestamps, see "String representations of datetime values" on page 101.

If *expression* is a timestamp with a time zone, or a valid string representation of a timestamp with a time zone, the result is determined from the UTC representation of the datetime value.

The result of the function is a large integer.

The result can be null; if the argument is null, the result is the null value.

Example 1: Find the number of seconds between midnight and 00:01:00, and midnight and 13:10:10. Assume that host variable *XTIME1* has a value of '00:01:00', and that *XTIME2* has a value of '13:10:10'.

```
SELECT MIDNIGHT_SECONDS(:XTIME1), MIDNIGHT_SECONDS(:XTIME2)
  FROM SYSIBM.SYSDUMMY1;
```

This example returns 60 and 47410. Because there are 60 seconds in a minute and 3600 seconds in an hour, 00:01:00 is 60 seconds after midnight ((60 * 1) + 0), and 13:10:10 is 47410 seconds ((3600 * 13) + (60 * 10) + 10).

Example 2: Find the number of seconds between midnight and 24:00:00, and midnight and 00:00:00.

```
SELECT MIDNIGHT_SECONDS('24:00:00'), MIDNIGHT_SECONDS('00:00:00')
  FROM SYSIBM.SYSDUMMY1;
```

This example returns 86400 and 0. Although these two values represent the same point in time, different values are returned.

Example 3: The following invocations of the MIDNIGHT_SECONDS function returns the same result:

```
SELECT MIDNIGHT_SECONDS('2003-01-02-20.10.05.123456'),
   MIDNIGHT_SECONDS('2003-01-02-12.10.05.123456-08:00'),
   MIDNIGHT_SECONDS('2003-01-03-05.10.05.123456+09:00')
  FROM SYSIBM.SYSDUMMY1;
```

For each invocation of the MIDNIGHT_SECONDS function in this SELECT statement, the result is 72605.

When the input argument contains a time zone, the result is determined from the UTC representation of the input value. The string representations of a timestamp with a time zone in the SELECT statement all have the same UTC representation: 2003-01-02-20.10.05.123456.

MIN

The MIN scalar function returns the minimum value in a set of values.

The schema is SYSIBM.

The arguments must be compatible. For more information on compatibility, refer to the compatibility matrix in Table 23 on page 121. All but the first argument can be parameter markers. There must be two or more arguments.

Each argument must be an expression that returns a value of any built-in data type other than a CLOB, DBCLOB, BLOB, ROWID, or XML.

Character string arguments and binary string arguments cannot have a length attribute greater than 32704, and graphic string arguments cannot have a length attribute greater than 16352.

The arguments are evaluated in the order in which they are specified. The result of the function is the minimum argument value.

The result can be null; if any argument is null, the result is the null value.

The selected argument is converted, if necessary, to the attributes of the result. The attributes of the result are determined using the "Rules for result data types" on page 144. If the MIN function has more than two arguments, the rules are applied to the first two arguments to determine a candidate result type. The rules are then applied to that candidate result type and the third argument to determine another candidate result type. This process continues until all arguments are analyzed and the final result type and CCSID is determined.

LEAST can be specified as a synonym for MIN.

Example 1: Assume the host variable *M1* is a DECIMAL(2,1) host variable with a value of 5.5, host variable *M2* is a DECIMAL(3,1) host variable with a value of 4.5, and host variable *M3* is a DECIMAL(3,2) host variable with a value of 6.25. The following function returns the value 4.5.

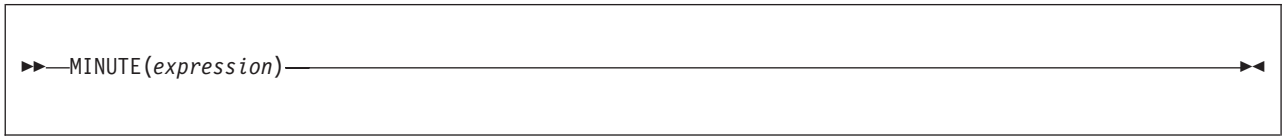
```
MIN(:M1,:M2,:M3)
```

Example 2: Assume the host variable *M1* is a CHAR(2) host variable with a value of 'AA', host variable *M2* is a CHAR(3) host variable with a value of 'AAA', and host variable *M3* is a CHAR(4) host variable with a value of 'AAAA'. The following function returns the value 'AA'.

```
MIN(:M1,:M2,:M3)
```

MINUTE

The MINUTE function returns the minute part of a value.



The schema is SYSIBM.

The argument must be an expression that returns a value of one of the following built-in data types: a time, a timestamp, a character string, a graphic string, or a numeric data type.

- If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a time or timestamp with an actual length of not greater than 255 bytes. For the valid formats of string representations of times and timestamps, see "String representations of datetime values" on page 101.
- If *expression* is a number, it must be a time or timestamp duration. For the valid formats of time and timestamp durations, see "Datetime operands" on page 147.

If *expression* is a timestamp with a time zone, or a valid string representation of a timestamp with a time zone, the result is determined from the UTC representation of the datetime value.

The result of the function is a large integer.

The result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

If the argument is a time, timestamp, or string representation of either, the result is the minute part of the value, which is an integer between 0 and 59.

If the argument is a time duration or timestamp duration, the result is the minute part of the value, which is an integer between -99 and 99. A nonzero result has the same sign as the argument.

If the argument contains a time zone, the result is the year part of the value expressed in UTC.

Example 1: Assume that a table named CLASSES contains one row for each scheduled class. Assume also that the class starting times are in the TIME column named STARTTM. Using these assumptions, select those rows in CLASSES that represent classes that start on the hour.

```
SELECT * FROM CLASSES
  WHERE MINUTE(STARTTM) = 0;
```

Example 2: The following invocations of the MINUTE function returns the same result:

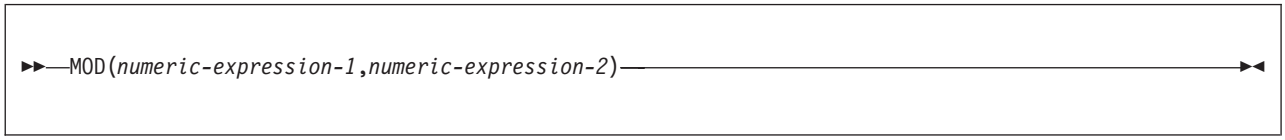
```
SELECT MINUTE('2003-01-02-20.10.05.123456'),
   MINUTE('2003-01-02-12.10.05.123456-08:00'),
   MINUTE('2003-01-03-05.10.05.123456+09:00')
  FROM SYSIBM.SYSDUMMY1;
```

For each invocation of the MINUTE function in this SELECT statement, the result is 2.

When the input argument contains a time zone, the result is determined from the UTC representation of the input value. The string representations of a timestamp with a time zone in the SELECT statement all have the same UTC representation: 2003-01-02-20.10.05.123456. The minute portion of the UTC representation is 10.

MOD

The MOD function divides the first argument by the second argument and returns the remainder.



The schema is SYSIBM.

The formula used to calculate the remainder is:

```
MOD(x,y) = x - FLOOR(x/y) * y
```

Where `x/y` is the truncated integer result of the division. The result is negative only if the first argument is negative.

Each argument must be an expression that returns a value of any built-in numeric data type.

The arguments can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34).

The result can be null; if any argument is null, the result is the null value.

The attributes of the result are based on the arguments as follows:

- If both arguments are large or small integers, the data type of the result is large integer.
- If both arguments are integers and at least one argument is a big integer, the data type of the result is big integer.
- If one argument is an integer and the other is a decimal, the data type of the result is decimal with the same precision and scale as the decimal argument.
- If both arguments are decimal, the data type of the result is decimal. The precision of the result is `min(`*p*`-`*s*`,`*p*`'-`*s*`')` + `max(`*s*`,`*s*`')`, and the scale of the result is `max(`*s*`,`*s*`')`, where the symbols *p* and *s* denote the precision and scale of the first argument, and the symbols *p'* and *s'* denote the precision and scale of the second argument.
- If one argument is a floating-point number, and the other is not a DECFLOAT, or both argument is a floating-point number, the data type of the result is double precision floating-point.

  The operation is performed in floating-point. If necessary, the operands are first converted to double precision floating-point numbers. For example, an operation that involves a floating-point number and either an integer or a decimal number is performed with a temporary copy of the integer or decimal number that has been converted to double precision floating-point. The result of a floating-point operation must be within the range of floating-point numbers.
- If either argument is a DECFLOAT, the data type of the result is DECFLOAT(34).

  If either argument is a special decimal floating point value, the general rules for arithmetic operations apply. See "General Arithmetic Operation Rules for DECFLOAT" on page 248 for more information.

If one argument is a DECFLOAT and the second argument is zero, the result is NaN and an invalid operation condition is returned.

Example: Assume that *M1* and *M2* are two host variables. Find the remainder of dividing *M1* by *M2*.

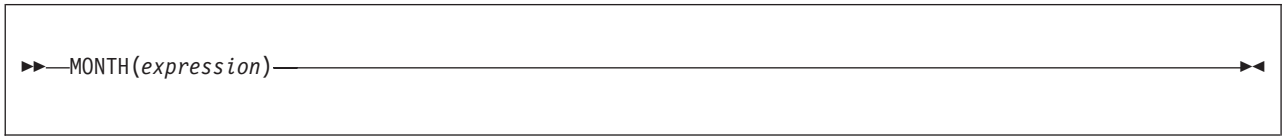
```
SELECT MOD(:M1,:M2)
  FROM SYSIBM.SYSDUMMY1;
```

The following table shows the result for this function for various values of *M1* and *M2*.

M1 **data type**	*M1* **value**	*M2* **data type**	*M2* **value**	**Result of `MOD(:M1,:M2)`**
INTEGER	5	INTEGER	2	1
INTEGER	5	DECIMAL(3,1)	2.2	0.6
INTEGER	5	DECIMAL(3,2)	2.20	0.60
DECIMAL(4,2)	5.50	DECIMAL(4,1)	2.0	1.50
DECFLOAT	1	DECFLOAT	-INFINITY	1
DECFLOAT	-0	DECFLOAT	INFINITY	-0
DECFLOAT	-0	DECFLOAT	-INFINITY	-0

MONTH

The MONTH function returns the month part of a value.



The schema is SYSIBM.

The argument must be an expression that returns one of the following built-in data types: a date, a timestamp, a character string, a graphic string, or a numeric data type.

- If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp with an actual length of not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 101.
- If *expression* is a number, it must be a date or timestamp duration. For the valid formats of date and timestamp durations, see "Datetime operands" on page 147.

If *expression* is a timestamp with a time zone, or a valid string representation of a timestamp with a time zone, the result is determined from the UTC representation of the datetime value.

The result of the function is a large integer.

The result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

If the argument is a date, timestamp, or string representation of either, the result is the month part of the value, which is an integer between 1 and 12.

If the argument is a date duration or timestamp duration, the result is the month part of the value, which is an integer between -99 and 99. A nonzero result has the same sign as the argument.

If the argument contains a time zone, the result is the year part of the value expressed in UTC.

Example 1: Select all rows in the sample table DSN8B10.EMP for employees who were born in May:

```
SELECT * FROM DSN8B10.EMP
  WHERE MONTH(BIRTHDATE) = 5;
```

Example 2: The following invocations of the MONTH function returns the same result:

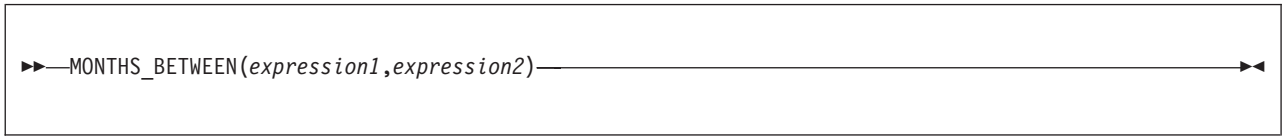
```
SELECT MONTH('2003-01-02-20.10.05.123456'),
   MONTH('2003-01-02-12.10.05.123456-08:00'),
   MONTH('2003-01-03-05.10.05.123456+09:00')
  FROM SYSIBM.SYSDUMMY1;
```

For each invocation of the MONTH function in this SELECT statement, the result is 1.

When the input argument contains a time zone, the result is determined from the UTC representation of the input value. The string representations of a timestamp with a time zone in the SELECT statement all have the same UTC representation: 2003-01-02-20.10.05.123456. The month portion of the UTC representation is 1.

MONTHS_BETWEEN

The MONTHS_BETWEEN function returns an estimate of the number of months between two arguments.



The schema is SYSIBM.

expression1 **or** *expression2*
Expressions that return a value of any of the following built-in data types: a date, a timestamp, a character string, or a graphic string. If either expression is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp with an actual length that is not greater than 255 bytes. A time zone in a string representation of a timestamp is ignored. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 101.

If *expression1* is a TIMESTAMP WITH TIME ZONE value, *expression1* is first cast to TIMESTAMP WITHOUT TIME ZONE with the same precision as *expression1*. If *expression2* is a TIMESTAMP WITH TIME ZONE value, *expression2* is first cast to TIMESTAMP WITHOUT TIME ZONE with the same precision as *expression2*.

If *expression1* represents a date that is later than *expression2*, the result is positive. If *expression1* represents a date that is earlier than *expression2*, the result is negative.

- If *expression1* and *expression2* represent dates or timestamps with the same day of the month, or both arguments represent the last day of their respective months, the result is a the whole number difference based on the year and month values, ignoring any time portions of timestamp arguments.
- Otherwise, the whole number part of the result is the difference based on the year and month values. The fractional part of the result is calculated from the remainder based on an assumption that every month has 31 days. If either argument represents a timestamp, the arguments are effectively processed as timestamps with maximum precision, and the time portions of these values are also considered when determining the result.

The result of the function is a DECIMAL(31,15).

The result can be null; if any argument is null, the result is the null value.

Examples 1: The following example calculates the months between two dates:

```
SELECT MONTHS_BETWEEN ('2008-01-17','2008-02-17')
    AS MONTHS_BETWEEN
    FROM SYSIBM.SYSDUMMY1;
```

The results of this statement are similar to the following results:

```
MONTHS_BETWEEN
---------------------------

-1.00000000000000
```

Examples 2: The following example calculates the months between two dates:

```
SELECT MONTHS_BETWEEN ('2008-02-20','2008-01-17')
    AS MONTHS_BETWEEN
    FROM SYSIBM.SYSDUMMY1;
```

The results of this statement are similar to the following results:

```
MONTHS_BETWEEN
--------------------------

1.096774193548387
```

Example 3: Calculate the number of months that project AD3100 will take. Assume that the start date is 1982-01-01 and the end date is 1983-02-01:

```
SELECT MONTHS_BETWEEN (PRENDATE, PRSDATE)
       FROM PROJECT
       WHERE PROJNO='AD3100';
```

The result is `13.000000000000000`.

Example 4: The following table illustrates the use of the MONTHS_BETWEEN function in certain situations:

Table 66. Additional examples using MONTHS_BETWEEN

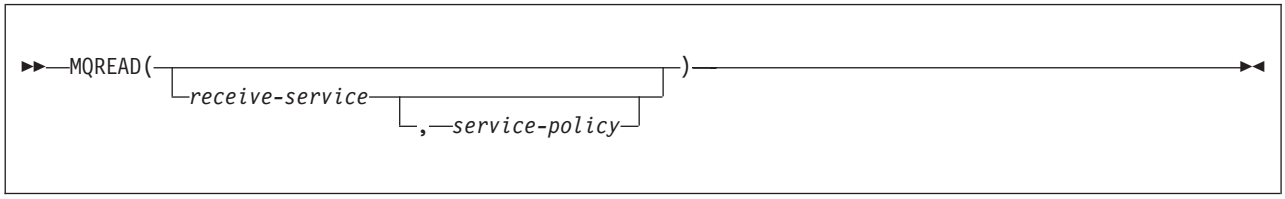
Value for *expression1*	**Value for** *expression2*	**Value returned by MONTHS_BETWEEN (***expression1***,***expression2***)**	**Value returned by ROUND (MONTHS_BETWEEN (***expression1***,***expression2***)*31,2)**
2005-02-02	2005-01-01	1.032258064516129	32.00
2007-11-01-09.00.00.00000	2007-12-07-14.30.12.12345	-1.200945386592741	-37.23
2007-12-13-09.40.30.00000	2007-11-13-08.40.30.00000	1.000000000000000[1]	31.00[1]
2007–03–15	2007–02–20	0.838709677419354[2]	26.00[2]
2008-02-29	2008-02-28-12.00.00	0.016129032258064	0.50
2008-03-29	2008-02-29	1.000000000000000	31.00
2008-03-30	2008-02-29	1.032258064516129	32.00
2008-03-31	2008-02-29	1.000000000000000[3]	31.00[3]

Notes:

1. The time difference is ignored because the day of the month is the same for both values.
2. The result is not 23 because, even though February has 28 days, the assumption is that all months have 31 days.
3. The result is not 33 because both dates are the last day of their respective month, and so the result is only based on the year and month portions.

MQREAD

The MQREAD function returns a message from a specified MQSeries location without removing the message from the queue.



The schema is DB2MQ.

The MQREAD function returns a message from the MQSeries location that is specified by *receive-service*, using the quality-of-service policy that is defined in *service-policy*. Performing this operation does not remove the message from the queue that is associated with *receive-service*, but instead returns the message at the beginning of the queue.

receive-service
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service point that is defined in the DB2MQ.MQSERVICE table. A service point is a logical end-point from which a message is sent or received. A service point definition includes the name of the MQSeries queue manager and the name of the queue. See *MQSeries Application Messaging Interface* for more details.

If *receive-service* is not specified or is the null value, DB2.DEFAULT.SERVICE is used.

service-policy
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service policy that is defined in the DB2MQ.MQPOLICY table. A service policy specifies a set of quality-of-service options that are to be applied to this messaging operation. These options include message priority and message persistence. See *MQSeries Application Messaging Interface* for more details.

If *service-policy* is not specified or is the null value, DB2.DEFAULT.POLICY is used.

The result of the function is a varying-length string with a length attribute of 4000. The result can be null. If no messages are available to be returned, the result is the null value.

The CCSID of the result is the system CCSID that was in effect at the time that the MQSeries function was installed into DB2.

Example 1: Retrieve the message at the beginning of the queue that is specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY).

```
SELECT MQREAD()
  FROM SYSIBM.SYSDUMMY1;
```

The message at the beginning of the queue specified by the default server and using the default policy is returned as VARCHAR(4000).

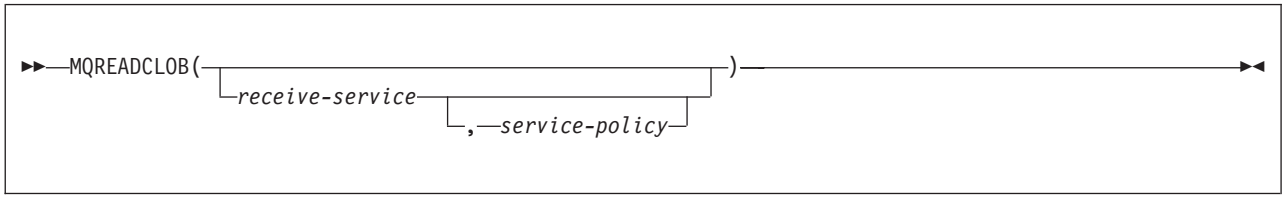
Example 2: Read the message from the beginning of the queue specified by the service MYSERVICE, using the default policy (DB2.DEFAULT.POLICY).

```
SELECT MQREAD('MYSERVICE')
  FROM SYSIBM.SYSDUMMY1;
```

The message at the beginning of the queue specified by MYSERVICE and using DB2.DEFAULT.POLICY is returned as VARCHAR(4000).

MQREADCLOB

The MQREADCLOB function returns a message from a specified MQSeries location without removing the message from the queue.



The schema is DB2MQ.

The MQREADCLOB function returns a message from the MQSeries location that is specified by *receive-service*, using the quality-of-service policy that is defined in *service-policy*. Performing this operation does not remove the message from the queue that is associated with *receive-service*, but instead returns the message at the beginning of the queue.

receive-service
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service point that is defined in the DB2MQ.MQSERVICE table. A service point is a logical end-point from which a message is sent or received. A service point definition includes the name of the MQSeries queue manager and the name of the queue. See *MQSeries Application Messaging Interface* for more details.

If *receive-service* is not specified or is the null value, DB2.DEFAULT.SERVICE is used.

service-policy
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service policy that is defined in the DB2MQ.MQPOLICY table. A service policy specifies a set of quality-of-service options that are to be applied to this messaging operation. These options include message priority and message persistence. See *MQSeries Application Messaging Interface* for more details.

If *service-policy* is not specified or is the null value, DB2.DEFAULT.POLICY is used.

The result of the function is a CLOB with a length attribute of 1 MB. The result can be null. If no messages are available to be returned, the result is the null value.

The CCSID of the result is the system CCSID that was in effect at the time that the MQSeries function was installed into DB2.

Example 1: Read the message from the beginning of the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY).

```
SELECT MQREADCLOB()
  FROM SYSIBM.SYSDUMMY1;
```

The message at the beginning of the queue specified by the default service and using the default policy is returned as a CLOB.

Example 2: Read the message from the beginning of the queue specified by the service MYSERVICE, using the default policy (DB2.DEFAULT.POLICY).

```
SELECT MQREADCLOB('MYSERVICE')
  FROM SYSIBM.SYSDUMMY1;
```

The message at the beginning of the queue specified by MYSERVICE and using the default policy is returned as a CLOB.

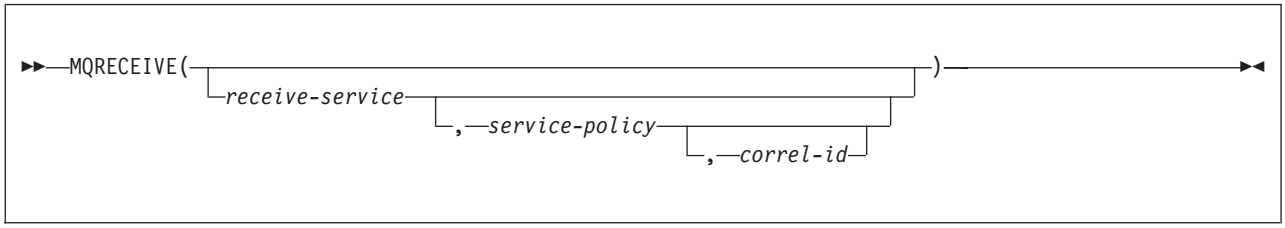
Example 3: Read the message from the beginning of the queue specified by the service MYSERVICE, using the policy MYPOLICY:

```
SELECT MQREADCLOB('MYSERVICE','MYPOLICY')
  FROM SYSIBM.SYSDUMMY1;
```

The message at the beginning of the queue specified by MYSERVICE and using the policy MYPOLICY is returned as a CLOB.

MQRECEIVE

The MQRECEIVE function returns a message from a specified MQSeries location and removes the message from the queue.



The schema is DB2MQ.

The MQRECEIVE function returns a message from the MQSeries location specified by *receive-service*, using the quality-of-service policy defined in *service-policy*. Performing this operation removes the message from the queue that is associated with *receive-service*.

receive-service
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service point that is defined in the DB2MQ.MQSERVICE table. A service point is a logical end-point from which a message is sent or received. A service point definition includes the name of the MQSeries queue manager and the name of the queue. See *MQSeries Application Messaging Interface* for more details.

If *receive-service* is not specified or is the null value, DB2.DEFAULT.SERVICE is used.

service-policy
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service policy that is defined in the DB2MQ.MQPOLICY table. A service policy specifies a set of quality-of-service options that are to be applied to this messaging operation. These options include message priority and message persistence. See *MQSeries Application Messaging Interface* for more details.

If *service-policy* is not specified or is the null value, DB2.DEFAULT.POLICY is used.

correl-id
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The expression must have an actual length that is no greater than 24 bytes. The value of the expression specifies the correlation identifier that is associated with this message. A correlation identifier is often specified in request-and-reply scenarios to associate requests with replies. Only those messages with a matching correlation identifier are returned.

A fixed length string with trailing blanks is considered a valid value. However, when the *correl-id* is specified on another request such as MQSEND, the *correl-id* must be specified the same to be recognized as a match. For example, specifying a value of 'test' for *correl-id* for this function does not match a *correl-id* value of 'test ' (with trailing blanks) specified earlier on an MQSEND request.

If *correl-id* is not specified, is an empty string, or is the null value, a correlation identifier is not used, and the message at the beginning of the queue is returned.

The result of the function is a varying-length string of length attribute of 4000. The result can be null. The result is null if no messages are available to return.

The CCSID of the result is the system CCSID that was in effect at the time that the MQSeries function was installed into DB2.

Example 1: Retrieve the message from beginning of the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY).

```
SELECT MQRECEIVE()
  FROM SYSIBM.SYSDUMMY1;
```

The message at the beginning of the queue is returned as VARCHAR(4000) and is deleted from the queue. The queue is specified by the default service and using the default policy.

Example 2: Retrieve the first message from the beginning of the queue specified by the service MYSERVICE, using the default policy, DB2.DEFAULT.POLICY.

```
SELECT MQRECEIVE('MYSERVICE')
  FROM SYSIBM.SYSDUMMY1;
```

The message at the beginning of the queue is returned as VARCHAR(4000) and is deleted from the queue. The queue is specified by the service MYSERVICE using the default policy, DB2.DEFAULT.POLICY.

Example 3: Retrieve the message from the beginning of the queue specified by the service MYSERVICE, using the policy MYPOLICY.

```
SELECT MQRECEIVE('MYSERVICE','MYPOLICY')
  FROM SYSIBM.SYSDUMMY1;
```

The message at the beginning of the queue is returned as VARCHAR(4000) and the message is deleted from the queue. The queue is specified by the service MYSERVICE using the policy MYPOLICY.

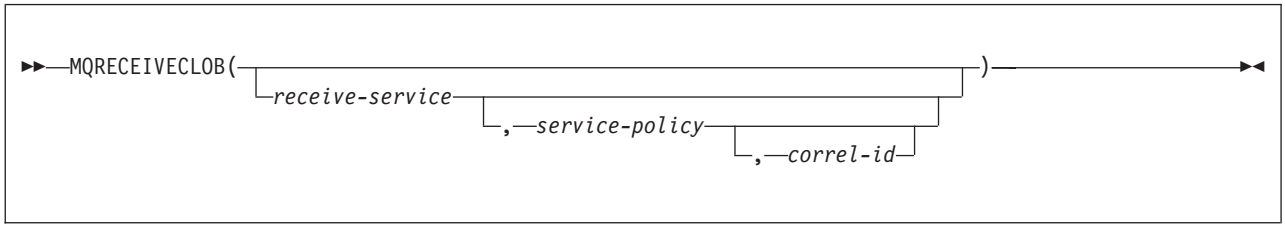
Example 4: Retrieve the first message with a correlation identifier that matches '1234' from the beginning of the queue specified by the service MYSERVICE, using the policy MYPOLICY.

```
SELECT MQRECEIVE('MYSERVICE','MYPOLICY','1234')
  FROM SYSIBM.SYSDUMMY1;
```

The first message with CORRELID of '1234' from the beginning of the queue is returned as VARCHAR(4000) and is deleted from the queue. The queue is specified by MYSERVICE and using MYPOLICY.

MQRECEIVECLOB

The MQRECEIVECLOB function returns a message from a specified MQSeries location and removes the message from the queue.



The schema is DB2MQ.

The MQRECEIVECLOB function returns a message from the MQSeries location that is specified by *receive-service*, using the quality-of-service policy that is defined in *service-policy*. Performing this operation removes the message from the queue that is associated with *receive-service*.

receive-service
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service point that is defined in the DB2MQ.MQSERVICE table. A service point is a logical end-point from which a message is sent or received. A service point definition includes the name of the MQSeries queue manager and the name of the queue. See *MQSeries Application Messaging Interface* for more details.

If *receive-service* is not specified or is the null value, DB2.DEFAULT.SERVICE is used.

service-policy
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service policy that is defined in the DB2MQ.MQPOLICY table. A service policy specifies a set of quality-of-service options that are to be applied to this messaging operation. These options include message priority and message persistence. See *MQSeries Application Messaging Interface* for more details.

If *service-policy* is not specified or is the null value, DB2.DEFAULT.POLICY is used.

correl-id
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The expression must have an actual length that is no greater than 24 bytes. The value of the expression specifies the correlation identifier that is associated with this message. A correlation identifier is often specified in request-and-reply scenarios to associate requests with replies. Only those messages with a matching correlation identifier are returned.

A fixed length string with trailing blanks is considered a valid value. However, when the *correl-id* is specified on another request such as MQSEND, the *correl-id* must be specified the same to be recognized as a match. For example, specifying a value of 'test' for *correl-id* for this function does not match a *correl-id* value of 'test ' (with trailing blanks) specified earlier on an MQSEND request.

If *correl-id* is not specified, is an empty string, or is the null value, a correlation identifier is not used, and the message at the beginning of the queue is returned.

The result of the function is a CLOB with a length attribute of 1 MB. The result can be null. If no messages are available to be returned, the result is the null value.

The CCSID of the result is the system CCSID that was in effect at the time that the MQSeries function was installed into DB2.

Example 1: Retrieve the message from the beginning of the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY).

```
SELECT MQRERECEIVECLOB()
  FROM SYSIBM.SYSDUMMY1;
```

The message at the beginning of the queue is returned as a CLOB and is deleted from the queue. The queue is specified by the default service and using the default policy.

Example 2: Retrieve the message from the beginning of the queue specified by the service MYSERVICE, using the policy (DB2.DEFAULT.POLICY).

```
SELECT MQRECEIVECLOB('MYSERVICE')
  FROM SYSIBM.SYSDUMMY1;
```

The message at the beginning of the queue is returned as a CLOB and is deleted from the queue. The queue is specified by MYSERVICE and using the default policy.

Example 3: Retrieve the message from the beginning of the queue specified by the service MYSERVICE, using the policy MYPOLICY.

```
SELECT MQRECEIVECLOB('MYSERVICE','MYPOLICY')
  FROM SYSIBM.SYSDUMMY1;
```

The message at the beginning of the queue is returned as a CLOB and is deleted from the queue. The queue is specified by MYSERVICE and using the policy MYPOLICY.

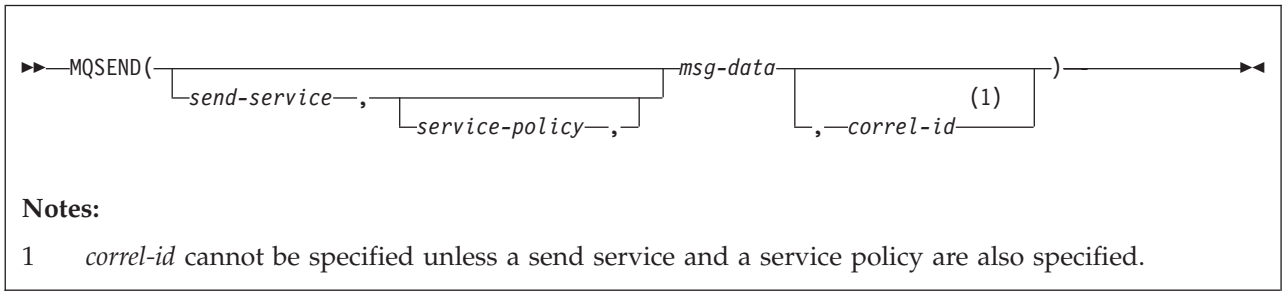
Example 4: Retrieve the first message from the beginning of the queue with a correlation identifier that matches '1234' from the queue specified by the service MYSERVICE, using the policy MYPOLICY.

```
SELECT MQRECEIVECLOB('MYSERVICE','MYPOLICY','1234')
  FROM SYSIBM.SYSDUMMY1;
```

The first message at the beginning of the queue with a correlation identifier with '1234' is returned as a CLOB and is deleted from the queue. The queue is specified by MYSERVICE and using the policy MYPOLICY.

MQSEND

The MQSEND function sends data to a specified MQSeries location, and returns a varying-length character string that indicates whether the function was successful or unsuccessful.



Notes:

1 *correl-id* cannot be specified unless a send service and a service policy are also specified.

The schema is DB2MQ.

The MQSEND function sends the data that is contained in *msg-data* to the MQSeries location that is specified by *send-service*, using the quality-of-service policy that is defined in *service-policy*. The returned value is '1' if the function was successful or '0' if unsuccessful.

send-service
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service point that is defined in the DB2MQ.MQSERVICE table. A service point is a logical end-point from which a message is sent or received. A service point definition includes the name of the MQSeries queue manager and the name of the queue. See *MQSeries Application Messaging Interface* for more details.

If *send-service* is not specified or is the null value, DB2.DEFAULT.SERVICE is used.

service-policy
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service policy that is defined in the DB2MQ.MQPOLICY table. A service policy specifies a set of quality-of-service options that are to be applied to this messaging operation. These options include message priority and message persistence. See *MQSeries Application Messaging Interface* for more details.

If *service-policy* is not specified or is the null value, DB2.DEFAULT.POLICY is used.

msg-data
An expression that returns a value that is a built-in character string data type. If the expression is a CLOB, the value must not be longer than 1 MB. Otherwise, the value must not be longer than 4000 bytes. The value of the

expression is the message data that is to be sent via MQSeries. A null value, an empty string, and a fixed length string with trailing blanks are all considered valid values.

correl-id
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The expression must have an actual length that is no greater than 24 bytes. The value of the expression specifies the correlation identifier that is associated with this message. A correlation identifier is often specified in request-and-reply scenarios to associate requests with replies. *correl-id* must not be specified unless *send-service* and *service-policy* are also specified.

A fixed length string with trailing blanks is considered a valid value. However, when the *correl-id* is specified on another request such as MQRECEIVE, the *correl-id* must be specified the same to be recognized as a match. For example, specifying a value of 'test' for *correl-id* on MQSEND does not match a *correl-id* value of 'test ' (with trailing blanks) specified subsequently on an MQRECEIVE request.

If *correl-id* is not specified, is an empty string, or is the null value, a correlation identifier is not sent.

The result of the function is a varying-length string with a length attribute of 1. The result is nullable, even though a null value is never returned. The result is '1' if the function is successful or '0' if unsuccessful.

The CCSID of the result is the system CCSID that was in effect at the time that the MQSeries function was installed into DB2.

Example 1: Send the string "Testing msg" to the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY) and no correlation identifier.

```
SELECT MQSEND('Testing msg')
  FROM SYSIBM.SYSDUMMY1;
```

The message is sent to the default service, using the default policy.

Example 2: Send the message 'Testing 345' to the service MYSERVICE, using the policy MYPOLICY, with no correlation identifier.

```
SELECT MQSEND('MYSERVICE','MYPOLICY','Testing 345')
  FROM SYSIBM.SYSDUMMY1;
```

The message 'Testing 345' is sent to the MYSERVICE service, using the policy MYPOLICY.

Example 3: Send the message 'Testing 123' to the service MYSERVICE, using the policy MYPOLICY and the correlation identifier 'TEST3'.

```
SELECT MQSEND('MYSERVICE','MYPOLICY','Testing 123','TEST3')
  FROM SYSIBM.SYSDUMMY1;
```

The message 'Testing 123' is sent to the service MYSERVICE, using the policy MYPOLICY and the correlation identifier "TEST3".

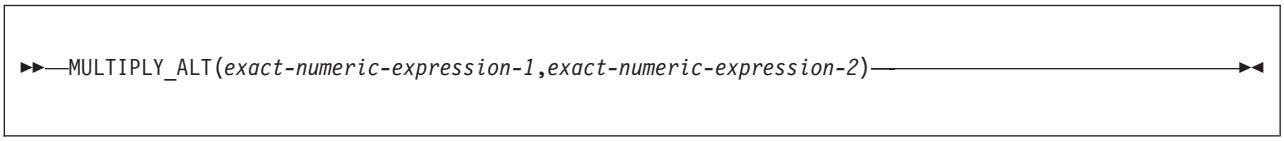
Example 4: Send the message 'Testing 901' to the service "MYSERVICE", using the default policy (DB2.DEFAULT.POLICY), and no correlation identifier.

```
SELECT MQSEND('MYSERVICE','Testing 901')
  FROM SYSIBM.SYSDUMMY1;
```

The message 'Testing 901' is sent to the service MYSERVICE, using the default policy (DB2.DEFAULT.POLICY).

MULTIPLY_ALT

The MULTIPLY_ALT scalar function returns the product of the two arguments. This function is an alternative to the multiplication operator and is especially useful when the sum of the precisions of the arguments exceeds 31.



The schema is SYSIBM.

Each argument must be an expression that returns the value of one of the following built-in numeric data types: DECIMAL, BIGINT, INTEGER, or SMALLINT.

The result of the function is a DECIMAL. The precision and scale of the result are determined as follows, using the symbols *p* and *s* to denote the precision and scale of the first argument, and the symbols *p'* and *s'* to denote the precision and scale of the second argument.

- The precision is `MIN(31, p+p')`
- The scale is:
  - 0 if the scale of both arguments is 0
  - `MIN(31, s+s')` if *p+p'* is less than or equal to 31
  - `MAX( MIN(3, s+s'), 31-(p-s+p'-s') )` if *p+p'* is greater than 31.

The result can be null; if any argument is null, the result is the null value.

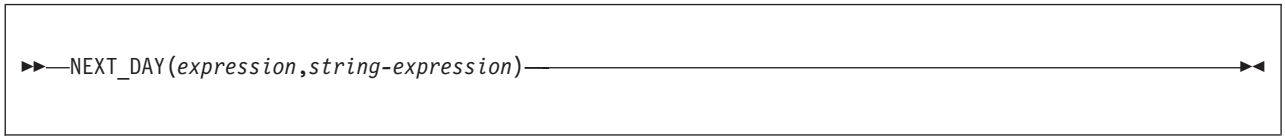
The MULTIPLY_ALT function is a better choice than the multiplication operator when performing decimal arithmetic where you want a scale of at least 3 and the sum of the precisions exceeds 31. In these cases, the internal computation is performed so that overflows are avoided and then assigned to the result type value using truncation for any loss of scale in the final result. Note that the possibility of overflow of the final result is still possible when the scale is 3.

The following table compares the result data types from the MULTIPLY_ALT function with the result data type of the multiplication operator when decimal data is used:

Type of Argument1	Type of Argument2	Result using MULTIPLY_ALT	Result using multiplication operator
DECIMAL(31,3)	DECIMAL(15,8)	DECIMAL(31,3)	DECIMAL(31,11)
DECIMAL(26,23)	DECIMAL(10,1)	DECIMAL(31,19)	DECIMAL(31,24)
DECIMAL(18,17)	DECIMAL(20,19)	DECIMAL(31,29)	DECIMAL(31,31)
DECIMAL(16,3)	DECIMAL(17,8)	DECIMAL(31,9)	DECIMAL(31,11)
DECIMAL(26,5)	DECIMAL(11,0)	DECIMAL(31,3)	DECIMAL(31,5)
DECIMAL(21,1)	DECIMAL(15,1)	DECIMAL(31,2)	DECIMAL(31,2)

NEXT_DAY

The NEXT_DAY function returns a datetime value that represents the first weekday, named by *string-expression*, that is later than the date in *expression*.



The schema is SYSIBM.

If *expression* is a timestamp or valid string representation of a timestamp, the timestamp value has the same hours, minutes, seconds, and partial seconds as *expression*. If *expression* is a date, or a valid string representation of a date, then the hours, minutes, seconds, and partial seconds value of the result is 0.

expression
An expression that returns one of the following built-in data types: a date, a timestamp, a character string, or a graphic string. If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp with an actual length of not greater than 255 bytes. A time zone in a string representation of a timestamp is ignored. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 101.

If *expression* does not have data type TIMESTAMP WITHOUT TIME ZONE, *expression* is cast as follows:

- If *expression* is a TIMESTAMP WITH TIME ZONE value, *expression* is cast to TIMESTAMP WITHOUT TIME ZONE, with the same precision as *expression*.
- Otherwise, *expression* is cast to TIMESTAMP(6) WITHOUT TIME ZONE.

string-expression
An expression that returns a built-in character or graphic string data type that is not a LOB. For portability across the platforms, the value should compare equal to the full name of a day of the week or should compare equal to the abbreviation of a day of the week. For example:

Day of week	Abbreviation
MONDAY	MON
TUESDAY	TUE
WEDNESDAY	WED
THURSDAY	THU
FRIDAY	FRI
SATURDAY	SAT
SUNDAY	SUN

The minimum length of the input value is the length of the abbreviation. Leading blanks must not be specified in *string-expression*. Trailing blanks are trimmed from *string-expression*. The resulting value is folded to uppercase. Any characters other than blank that immediately follow a valid abbreviation are ignored.

If *expression* is a timestamp, the result is a TIMESTAMP WITHOUT TIME ZONE value with the same precision as *expression*. If *expression* is DATE, the result is a DATE value. Otherwise, the result is a TIMESTAMP(6) WITHOUT TIME ZONE value.

Any hours, minutes, seconds, or fractional seconds information that is included in *expression* is not changed by the function. If *expression* is a string that represents a date, the time information in the resulting timestamp value is all set to zero.

The result can be null; if any argument is null, the result is the null value.

The result CCSID is the appropriate CCSID of the argument encoding scheme and the result subtype is the appropriate subtype of the CCSID.

Example 1: Set the host variable NEXTDAY with a timestamp for the date of the Tuesday that follows April 24, 2007.

```
SET :NEXTDAY = NEXT_DAY(TIMESTAMP '2007-04-24-00.00.00.000000', 'TUESDAY');
```

The host variable NEXTDAY is set with the value of '2007-05-01-00.00.00.000000', since April, 24, 200 is itself a Tuesday'.

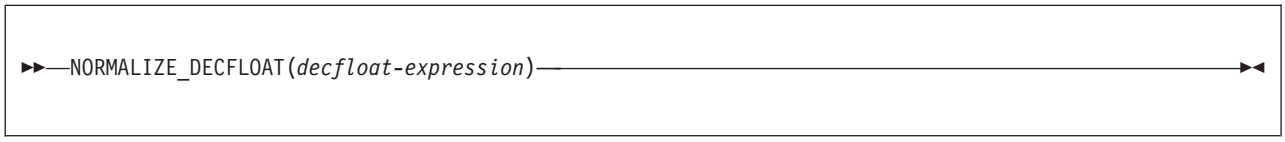
Example 2: Set the host variable vNEXTDAY with the date of the first Monday in May, 2007. Assume the host variable vDAYOFWEEK = 'MON':

```
SET :vNEXTDAY = NEXT_DAY(LAST_DAY(CURRENT_DATE),:vDAYOFWEEK);
```

The host variable vNEXTDAY is set with the value of '2007-05-07', assuming that the value of the CURRENT_DATE special register is '2007-04-24'.

NORMALIZE_DECFLOAT

The NORMALIZE_DECFLOAT function returns a DECFLOAT value that is the result of the argument, set to its simplest form. That is, a non-zero number that has any trailing zeros in the coefficient has those zeros removed by dividing the coefficient by the appropriate power of ten and adjusting the exponent accordingly. A zero has its exponent set to 0.



The schema is SYSIBM.

decfloat-expression
The argument must be an expression that returns a DECFLOAT value.

decfloat-expression can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34).

If the argument is a special decimal floating point value then the general rules for arithmetic operations apply. See "General Arithmetic Operation Rules for DECFLOAT" on page 248 for more information.

The result of the function is a DECFLOAT(16) value if the data type of *decfloat-expression* is DECFLOAT(16). Otherwise, the result of the function is a DECFLOAT(34) value.

The result can be null; if the argument is null, the result is the null value.

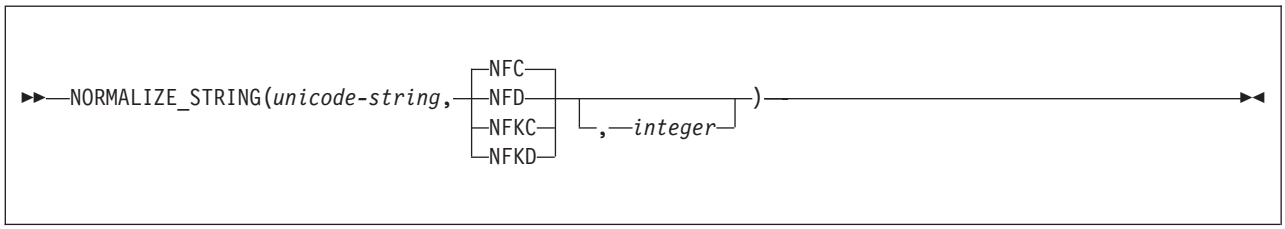
Examples: The following examples show the result of using the NORMALIZE_DECFLOAT function on various DECFLOAT values:

```
NORMALIZE_DECFLOAT(DECFLOAT(2.1))         = 2.1
NORMALIZE_DECFLOAT(DECFLOAT(-2.0))        = -2
NORMALIZE_DECFLOAT(DECFLOAT(1.200))       = 1.2
NORMALIZE_DECFLOAT(DECFLOAT(-120))        = -1.2E+2
NORMALIZE_DECFLOAT(DECFLOAT(120.00))      = 1.2E+2
NORMALIZE_DECFLOAT(DECFLOAT(0.00))        = 0
NORMALIZE_DECFLOAT(-NAN)                  = -NAN
NORMALIZE_DECFLOAT(-INFINITY)             = -INFINITY
```

NORMALIZE_STRING

The NORMALIZE_STRING function takes a Unicode string argument and returns a normalized string that can be used for comparison.

The NORMALIZE_STRING function can convert two strings that look the same (such as Å, which can be encoded in UTF-16 as X'00C5' and as X'0041030a') but might not be encoded using the same Unicode code point, to a normalized form that can be compared.



The schema is SYSIBM.

unicode_string
An expression that returns a value of a built-in character string or graphic string data type that is either Unicode UTF-8 or Unicode UTF-16, and is not a LOB. The CAST specification can be used to convert ASCII or EBCDIC data to Unicode for use with this function.

NFC, NFD, NFKC, or NFKD
Specifies the normalized form:
NFC Canonical Decomposition followed by Canonical Composition
NFD Canonical Decomposition
NFKC Compatibility Decomposition followed by Canonical Composition
NFKD Compatibility Decomposition

integer
The length attribute, in bytes if the string is a character string, or in double byte code points if the string is a graphic string, for the resulting variable length string. The value must be an integer between 1 and 32704 if the source string is character, or 16352 if the source string is graphic.

The result of the function is a varying length string with a data type that depends on the data type of *unicode-string*:

- VARCHAR if *unicode-string* is CHAR or VARCHAR
- VARGRAPHIC if *unicode-string* is GRAPHIC or VARGRAPHIC

The CCSID of the result is the same as the CCSID of *unicode-string*.

The length attribute of the result depends on whether *integer* is specified. If *integer* is specified, the length attribute of the result is *integer* bytes or double byte code points. If *integer* is not specified, the length attribute of the result is `MIN(3*n,32704)` for character strings, or `MIN(3*n,16352)` for graphic strings, where *n* is the length attribute of the source.

The result can be null; if the first argument is null, the result is the null value.

Example 1: In the following example, "ábc" is normalized to normalization form NFC:

```
SET :hv1 = NORMALIZE_STRING('ábc',NFC) -- x'0061030100620063'
```

hv1 is set to 'ábc' -- X'00E100620063'. Using normalization form NFC, the two code-point sequence X'00610301', which represents the character 'á', is normalized to X'00E1' which is also the pre-composed equivalent of X'00610301'.

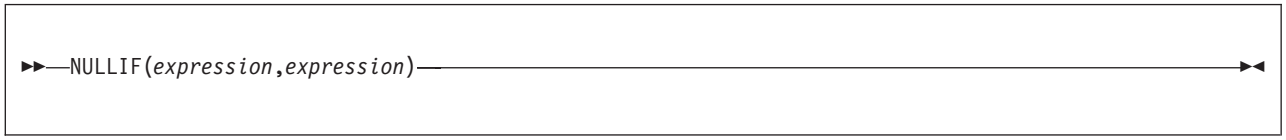
Example 2: In the following example, "ábc" is normalized to normalization form NFD.

```
SET :hv1 = NORMALIZE_STRING('ábc',NFD) -- x'00E100620063'
```

hv1 is set to 'ábc' -- X'0061030100620063'. Using normalization form NFD, the code point X'00E1' is decomposed into the two code-point sequence X'00610301', which consists of the Latin lower case letter A and the combining acute accent character.

NULLIF

The NULLIF function returns the null value if the two arguments are equal; otherwise, it returns the value of the first argument.



The schema is SYSIBM.

The two arguments must be compatible. The arguments can be of either a built-in or user-defined distinct type. Neither argument can be a BLOB, CLOB, DBCLOB, or XML. Character-string and graphic-string arguments are compatible with datetime values. For more information on compatibility, refer to the compatibility matrix in Table 23 on page 121.

If there are any mixed character string or graphic string and numeric arguments, the string value is implicitly cast to a DECFLOAT(34) value.

The attributes of the result are the attributes of the first argument.

The result of using `NULLIF(e1,e2)` is the same as using the CASE expression:

```
CASE WHEN e1=e2 THEN NULL ELSE e1 END
```

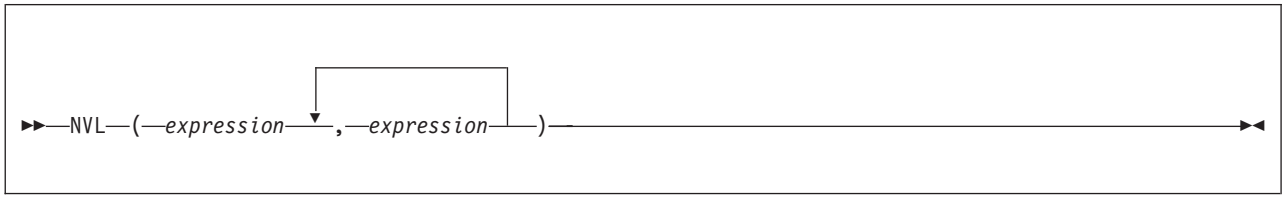
When `e1=e2` evaluates to unknown because one or both arguments is null, CASE expressions consider the evaluation not true. In this case, NULLIF returns the value of the first argument.

Example: Assume that host variables *PROFIT*, *CASH*, and *LOSSES* have decimal data types with the values of 4500.00, 500.00, and 5000.00 respectively. The following function returns a null value:

```
NULLIF (:PROFIT + :CASH , :LOSSES)
```

NVL

The NVL function returns the first argument that is not null.

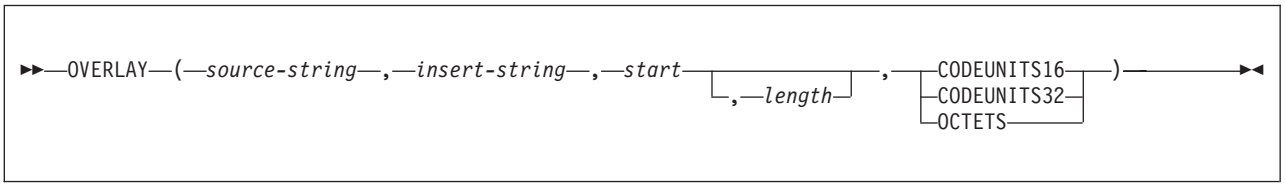


The schema is SYSIBM.

The NVL function is a synonym for the COALESCE function.

OVERLAY

The OVERLAY function returns a string that is composed of one argument that is inserted into another argument at the same position where some number of bytes have been deleted.



The schema is SYSIBM.

The OVERLAY function returns a string where a substring of *length*, beginning at *start* has been deleted from *source-string*, and where *insert-string* has been inserted into *source-string* beginning at *start*. If the value of *start* plus *length* is greater than the length of *source-string*, the substring that is deleted is from *start* to the end of *source-string*.

If the length of the result string exceeds the maximum for the return type, an error is returned.

The OVERLAY function is identical to the INSERT function, except that the length argument is optional.

source-string
An expression that specifies the source string. The expression must return a value that is a built-in character string, graphic string, or binary string data type that is not a LOB.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.The actual length of the string must be greater than or equal to 1 byte and less than or equal to 32704 bytes.

insert-string
An expression that specifies the string that is inserted into *source-string*, starting at the position that is identified by *start*. *insert-string* must return a value that is a built-in character string, graphic string, or binary string data type that is not a LOB. *source-string* and *insert-string* must have compatible data types.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.

start
An expression that returns an integer. The integer specifies the starting point within the source string where the deletion of bytes and the insertion of another string is to begin. The value of the integer must be in the range of 1 to the length of *source-string* plus one. If OCTETS is specified and the result is graphic data, the value must be an odd value between 1 and twice the length of *source-string* plus one.

The argument can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34) which is then assigned to an INTEGER value.

length
An expression that specifies the length of the string to replace in *source-string*

starting at *start*. *length* must be an expression that returns a value of the built-in INTEGER data type. *length* is expressed in the string unit specified, and the value must be in the range of 0 to the length of *source-string*. If OCTETS is specified and the result is graphic data, *length* must be even and be between 0 and twice the length of *source-string*. Not specifying *length* is equivalent to specifying a value of 1, except when OCTETS is specified and the result is graphic data, in which case, not specifying *length* is equivalent to specifying a value of 2.

The argument can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34) which is then assigned to an INTEGER value.

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the units that are used to express *start* and *length* in the result. If *source-string* is a character string that is defined as bit data, CODEUNITS16 and CODEUNITS32 cannot be specified. If *source-string* is a graphic string, OCTETS cannot be specified. If *source-string* is a binary string, CODEUNITS16, CODEUNITS32, and OCTETS cannot be specified.

If a string unit is not explicitly specified, the data type of the result determines the unit that is used. If the result is a graphic string, a string unit is two bytes. For ASCII and EBCDIC data, this corresponds to a double byte character. For Unicode, this corresponds to a UTF-16 code point. Otherwise, a string unit is a byte.

CODEUNITS16
Specifies that *start* and *length* are expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32
Specifies that *start* and *length* are expressed in terms of 32-bit UTF-32 code units.

OCTETS
Specifies that *start* and *length* are expressed in terms of bytes.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String unit specifications" on page 87. *length* must be an even number if *source-string* is graphic data and OCTETS is specified

If *source-string* and *insert-string* have different CCSID sets, *insert-string* (the string to be inserted) is converted to the CCSID of *source-string* (the source string).

The encoding scheme of the result is the same as *source-string*. The data type of the result of the function depends on the data type of *source-string* and *insert-string*:

- VARCHAR if *source-string* is a character string. The CCSID of the result depends on the arguments:
  - If either *source-string* or *insert-string* is character bit data, the result is bit data.
  - If both *source-string* and *insert-string* are SBCS:
    - If both *source-string* and *insert-string* are SBCS Unicode data, the CCSID of the result is the CCSID for SBCS Unicode data.
    - If *source-string* is SBCS Unicode data and *insert-string* is not SBCS Unicode data, the CCSID of the result is the mixed CCSID for Unicode data.
    - Otherwise, the CCSID of the result is the same as the CCSID of *source-string*.

  - Otherwise, the CCSID of the result is the mixed CCSID that corresponds to the CCSID of *source-string*. However, if the input is EBCDIC or ASCII and there is no corresponding system CCSID for mixed, the CCSID of the result is the CCSID of *source-string*.
- VARGRAPHIC if *source-string* is a graphic. The CCSID of the result is the same as the CCSID of *source-string*.
- VARBINARY if *source-string* and *insert-string* are both binary strings.

The length attribute of the result depends on the arguments:

- If *start* and *length* are constants, the length attribute of the result is:

  ```
  L1 - MIN((L1 - V2 + 1), V3) + L4
  ```

  where:
  L1 is the length attribute of *source-string*
  V2 is the value of *start*
  V3 is the value of *length*
  L4 is the length attribute of *insert-string*
- Otherwise, the length attribute of the result is the length attribute of *source-string* plus the length attribute of *insert-string*. In this case, the length attribute of *source-string* plus the length attribute of *insert-string* must not exceed 32704 for a VARCHAR result or 16352 for a VARGRAPHIC result.

If CODEUNITS16 or CODEUNITS32 is specified, the insert operation is performed on a Unicode version of the data. If needed, the data is converted to an intermediate form in order to evaluate the function. If an intermediate form is used, the actual length of the result depends on the original data (*source-string* and *insert-string*), and the representation of that data in Unicode. See "Determining the length attribute of the final result" on page 90 for more information on how to calculate the length attribute of the result string.

If CODEUNITS16 or CODEUNITS32 are not specified, the actual length of the result is:

```
A1 - MIN((A1 - V2 + 1), V3) + A4
```

where:
A1 is the actual length of *source-string*
V2 is the value of *start*
V3 is the value of *length*
A4 is the actual length of *insert-string*

If the actual length of the result string exceeds the maximum for the return data type, an error occurs.

The result can be null; if any argument is null, the result is the null value.

Example 1: The following example shows how the string 'INSERTING' can be changed into other strings. The use of the CHAR function limits the length of the resulting string to 10 bytes.

```
SELECT CHAR(OVERLAY('INSERTING','IS',4,2,OCTETS),10),
       CHAR(OVERLAY('INSERTING','IS',4,0,OCTETS),10),
       CHAR(OVERLAY('INSERTING','',4,2,OCTETS),10)
  FROM SYSIBM.SYSDUMMY1;
```

This example returns 'INSISTING ', 'INSISERTIN', and 'INSTING '

Example 2: Use the OVERLAY function to insert the character 'C' into the Unicode string '&N~AB', where '&' is the character for the musical symbol, G CLEF, and '~' is the character for combining tilde. The following table shows the Unicode string in different Unicode encoding forms:

Unicode format	&	N	~	A	B
UTF-8	X'F09D849E'	X'4E'	X'CC83'	X'41'	X'42'
UTF-16	X'D834DD1E'	X'004E'	X'0303'	X'0041'	X'0042'

Assume the host variable *UTF8_VAR* contains the UTF-8 representation of '&N~AB', and *UTF16_VAR* contains the UTF-16 representation of '&N~AB'. Then the following SELECT statement is run:

```
SELECT OVERLAY (:UTF8_VAR, 'C', 1, CODEUNITS16),
       OVERLAY (:UTF8_VAR, 'C', 1, CODEUNITS32),
       OVERLAY (:UTF8_VAR, 'C', 1, OCTETS)
   FROM SYSIBM.SYSDUMMY1
```

This statement returns the following values:

```
C N~AB
CN~AB
C?N~AB  -- ? is the invalid UTF-8 sequence X'9D849E'
```

Assume that the previous SELECT statement was not run, but the following SELECT statement is run:

```
SELECT OVERLAY (:UTF8_VAR, 'C', 5, CODEUNITS16),
       OVERLAY (:UTF8_VAR, 'C', 5, CODEUNITS32),
       OVERLAY (:UTF8_VAR, 'C', 5, OCTETS)
   FROM SYSIBM.SYSDUMMY1;
```

This statement returns the values:

```
&N~CB
&N~AC
&C~AB
```

Assume that the previous SELECT statement was not run, but the following SELECT statement is run:

```
 SELECT OVERLAY (:UTF16_VAR, 'C', 1, CODEUNITS16),
        OVERLAY (:UTF16_VAR, 'C', 1, CODEUNITS32)
    FROM SYSIBM.SYSDUMMY1;
```

This statement returns the values:

```
C?N~AB
CN~AB
```

Assume that the previous SELECT statement was not run, but the following SELECT statement is run:

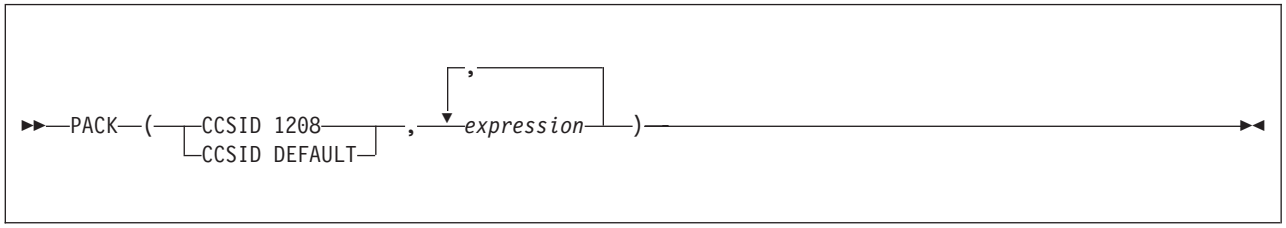
```
SELECT OVERLAY (:UTF16_VAR, 'C', 5, CODEUNITS16),
       OVERLAY (:UTF16_VAR, 'C', 5, CODEUNITS32),
   FROM SYSIBM.SYSDUMMY1;
```

This statement returns the values:

```
&N~CB
&N~AC
```

PACK

The PACK function returns a binary string value that contains a data type array and a packed representation of each non-null *expression* argument.



The schema is SYSIBM.

CCSID 1208
Specifies that CCSID 1208 is used to encode character string values.

CCSID DEFAULT
Specifies that character strings are to be packed in their original encoding, as-is, without CCSID conversion.

expression
An expression that returns a value to be encoded in the result string. The *expression* must be a built-in data type that is not DECFLOAT, GRAPHIC, VARGRAPHIC, ROWID, a LOB, XML, or a character string defined as FOR BIT DATA.

The result of the PACK function is a binary string that is constructed from the following items:

- A flag byte that is reserved for future use
- A 2-byte integer value that indicates the number of arguments encoded in the resulting string
- The data type array that contains an element with data type information for each of the encoded arguments
- The encoded values for the *expression* arguments in the order as specified in the function invocation.

The resulting binary string is formatted as follows:

2-byte length	Flag byte	Number of items	Data type array	Encoded data values
VARBINARY length	VARBINARY data			

The data type array includes an element for each *expression* argument in the same order as specified in the function invocation. Each array element contains a 2-byte SQLTYPE value that indicates the data type of the corresponding expression. When the SQLTYPE value is an odd number, the corresponding expression represents a null value and the value is not encoded in the resulting string. When the SQLTYPE value is an even number, the resulting string contains an encoded representation of the value depending on the data type. The following table describes the data types:

Table 67. Data types for the expression of the PACK function

Data type of expression	Description of the encoded representation of the value in the resulting string
SMALLINT, INTEGER, or BIGINT	The value of expression as a 16-bit signed binary integer, 32-bit signed binary integer, or 64-bit signed binary integer depending on the data type
decimal(*p*,*s*)[1]	A sequence of 1-byte precision *p*, 1-byte scale *s*, and (*p*+2)/2 bytes of the signed packed-decimal number
real[2] or double[3]	The value of *expression* as a 64-bit IEEE floating-point format
CHAR or VARCHAR	A sequence of the 2-byte CCSID of the string encoding, followed by the 2-byte length of the string and then the argument data in the specified CCSID encoding
BINARY or VARBINARY	A sequence of: 2-byte length of the string, followed by the argument data
DATE	A 4-byte unsigned packed-decimal number representation of the date in the form of YYYYMMDD
TIME	A 3-byte unsigned packed-decimal number representation of the time in the form of HHMMSS
TIMESTAMP(*p*) WITHOUT TIME ZONE	A sequence of a 2-byte unsigned binary integer value of the precision *p*, followed by 7+ (*p*+1)/2 bytes of an unsigned packed-decimal number representation of the timestamp in the form of YYYYMMDDHHMMSSNN, where NN is zero to six bytes of the fractional seconds, depending on the precision *p*
TIMESTAMP(p) WITH TIME ZONE	A sequence of a 2-byte unsigned binary integer value of the precision *p*, followed by 7+ (*p*+1)/2 bytes of an unsigned packed-decimal number representation of the timestamp in the form of YYYYMMDDHHMMSSNN, where NN is zero to six bytes of the fractional seconds, depending on the precision *p*, and then followed by 2 bytes of an unsigned packed-decimal number representation of the time zone (with high order bit set for negative time zone value)

Note: The data types in lower case are defined as follows:

1. **decimal** = DECIMAL(*p*,*s*) or NUMERIC(*p*,*s*)
2. **real** = REAL or FLOAT(*n*) where *n* is the specification for a single precision floating point
3. **double** = DOUBLE, DOUBLE PRECISION, FLOAT or FLOAT(*n*) where *n* is the specification for a double precision floating point

The synonyms for the data types, in either long or short form, are considered the same as those that are listed.

All numeric data is represented in big endian format.

The result of the function is VARBINARY. The length attribute of the result is MIN (32704, the length of the header + length of data type array + SUM(maximum lengths of encoded expression values)). The result cannot be null.

Example 1: The following statement shows that the VARCHAR, DATE, and DOUBLE values are packed into a binary string, and the string is then returned to the application:

```
SELECT PACK(CCSID 1208, 'Alina', DATE'1977-08-01', DOUBLE(0.5))
  FROM SYSIBM.SYSDUMMYU;
```

The statement returns a VARBINARY string with the following content (The result is displayed in hexadecimal format and includes space separators for readability. The actual result is not in hexadecimal format and does not include any space separators):

```
00 0003 01C4 0180 01E0 04B80005416C696E61 19770801 3FE0000000000000
```

Note that the character string 'Alina' is in UTF-8 (CCSID 1208) format regardless of the string's original encoding because of the CCSID 1208 specification in the PACK invocation.

The resulting string is VARBINARY(30). The length attribute of 30 is determined by the following elements:

1 (flag byte)
+2 (size of number of items)
+2*3 (2-byte data type times number of items)
+2 (CCSID) + 2 (length) + 5 (VARCHAR(5) data length)
+4 (DATE data length)
+8 (DOUBLE length)

The actual length of the result is also 30.

Example 2: The following statement shows that when NULL values are packed into a binary string, they do not occupy any space in the encoded values portion of the result:

```
SELECT PACK(CCSID 1208, '', CAST(NULL AS TIME),
       CAST('Bridget' AS VARCHAR(20) CCSID EBCDIC))
  FROM SYSIBM.SYSDUMMYU;
```

The statement returns a VARBINARY string with the following content (The result is displayed in hexadecimal format and includes space separators for readability. The actual result is not in hexadecimal format and does not include any space separators):

```
00 0003 01C4 0185 01C4 04B80000 00250007C2D9C9C4C7C5E3
```

Note that the character strings '' (empty string) and 'BRIDGET' are packed in their original CCSID 1208 and CCSID 37 format accordingly because of the CCSID DEFAULT specification in the PACK invocation.

The resulting string is VARBINARY(40). The length attribute of 40 is determined by the following elements:

1 (flag byte)
+2 (size of number of items)
+2*3 (2-byte data type times number of items)
+2 (CCSID) + 2 (length) + 0 (empty string data length)
+3 (TIME data length)
+2 (CCSID) + 2 (length) + 20 (VARCHAR(20) max length)

The actual length of the resulting string is 24, which is determined by the following elements

1 (flag byte)
+2 (size of number of items)
+2*3 (2-byte data type times number of items)

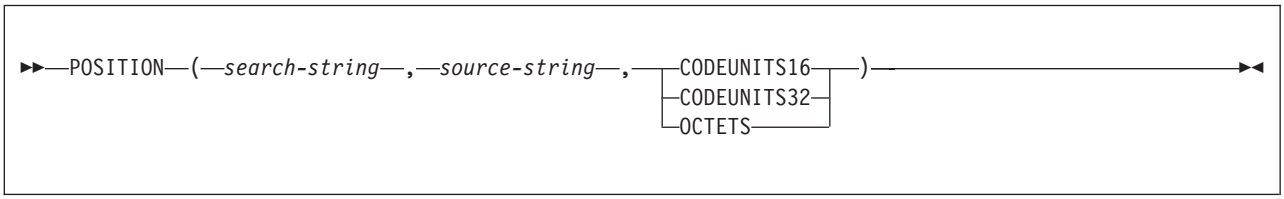
+2 (CCSID) + 2 (length) + 0 (empty string data length)
+0 (NULL)
+2 (CCSID) + 2 (length) + 7 (VARCHAR(20) actual length)

Related reference:

"SQLTYPE and SQLLEN" on page 2090
"PACK" on page 562

POSITION

The POSITION function returns the position of the first occurrence of an argument within another argument, where the position is expressed in terms of the string units that are specified.



The schema is SYSIBM.

If *search-string* is not found and neither argument is null, the result is 0. If *search-string* is found, the result is a number from 1 to the actual length of *source-string*, expressed in the units that are explicitly specified.

search-string
An expression that specifies the string for which to search. *search-string* must return a value that is any built-in string data type with an actual length that is no greater than 4000 bytes.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.The expression can be specified by any of the following items:

- A constant
- A special register
- A variable
- A scalar function whose arguments are any of the above (although nested function invocations cannot be used)
- An array element specification
- An expression that concatenates (using CONCAT or ||) any of the above
- A CAST specification whose arguments are any of the above
- A column name

These rules are similar to those that are described for *pattern-expression* for the LIKE predicate.

source-string
An expression that specifies the source string in which the search is to take place. *source-string* must return a value that is any built-in string data type.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.The expression can be specified by any of the following items:

- A constant
- A special register
- A variable
- A scalar function whose arguments are any of the above (though nested function invocations cannot be used)
- A column name
- An array element specification
- A CAST specification whose arguments are any of the above
- An expression that concatenates (using CONCAT or ||) any of the above

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the string unit that is used to express the result. If *source-string* is a character string that is defined as bit data, CODEUNITS16, or CODEUNITS32 cannot be specified. If *source-string* is a graphic string, OCTETS cannot be specified.

CODEUNITS16
Specifies that the result is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32
Specifies that the result is expressed in terms of 32-bit UTF-32 code units.

OCTETS
Specifies that the result is expressed in terms of bytes.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String unit specifications" on page 87.

The first and second arguments must have compatible string types. For more information on compatibility, see "Conversion rules for operations that combine strings".

If the search string and source string have different CCSID sets, then the *search-string* is converted to the CCSID set of the source string. If either CODEUNITS16 or CODEUNITS32 is specified, the function might be evaluated on a temporary copy of the data in Unicode.

The strings can contain mixed data. If OCTETS is specified:

- For ASCII data, if the search string or source string contains mixed data, the search string is found only if the same combination of single-byte and double-byte characters are found in the source string in exactly the same positions.
- For EBCDIC data, if the search string or source string contains mixed data, the search string is found only if any shift-in or shift-out characters are found in the source string in exactly the same positions, ignoring any redundant shift characters.
- For UTF-8 data, if the search string or source string contains mixed data, the search string is found only if the same combination of single-byte and multi-byte characters are found in the source string in exactly the same position.

The result of the function is a large integer. The POSITION function accepts mixed data strings.

The result can be null; if any argument is null, the result is the null value.

When the POSITION function is invoked with OCTETS, the function operates on a strict byte-count basis without regard to single-byte or double-byte characters.

If the CCSID of the search string is different than the CCSID of the source string, it is converted to the CCSID of the source string.

The value of the result is determined by applying these rules in the order in which they appear:

- If *search-string* has a length of zero, the result is 1.
- If *source-string* has a length of zero, the result is 0.

- If the value of *search-string* is equal to an identical length of substring of contiguous positions within the value of *source-string*, the result is the starting position of the first such substring within the source string value.
- Otherwise, the result is 0. This includes the case where *search-string* is longer than *source-string*.

Example1: Select the RECEIVED column, the SUBJECT column, and the starting position of the string 'GOOD BEER' within the NOTE_TEXT column for all rows in the IN_TRAY table that contain that string.

```
SELECT RECEIVED, SUBJECT, POSITION('GOOD BEER', NOTE_TEXT, OCTETS)
  FROM IN_TRAY
  WHERE POSITION('GOOD BEER', NOTE_TEXT, OCTETS) <> 0;
```

Example 2: Find the position of the character 'ß' in the string 'Jürgen lives on Hegelstraße', and set the host variable *LOCATION* with the position, as measured in CODEUNITS32 units, within the string.

```
SET :LOCATION = POSITION('ß','Jürgen lives on Hegelstraße',CODEUNITS32);
```

The value of host variable *LOCATION* is set to 27.

Example 3: Find the position of the character 'ß' in the string 'Jürgen lives on Hegelstraße', and set the host variable *LOCATION* with the position, as measured in OCTETS, within the string.

```
SET :LOCATION = POSITION('ß','Jürgen lives on Hegelstraße',OCTETS);
```

The value of host variable *LOCATION* is set to 28.

Related reference:

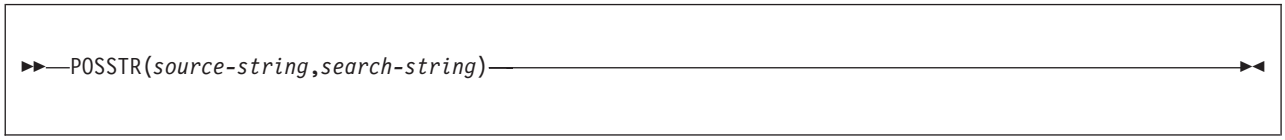
"LOCATE" on page 510

"LOCATE_IN_STRING" on page 513

"POSSTR" on page 569

POSSTR

The POSSTR function returns the position of the first occurrence of an argument within another argument.



The schema is SYSIBM.

If *search-string* is not found and neither argument is null, the result is 0. If *search-string* is found, the result is a number from 1 to the actual length of *source-string*.

source-string
An expression that specifies the source string in which the search is to take place. *source-string* must return a value that is a built-in character string data type, graphic string data type, or binary string data type.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.The expression can be specified by any of the following items:
- A constant
- A special register
- A variable
- A scalar function whose arguments are any of the above (though nested function invocations cannot be used)
- A column name
- An array element specification
- A CAST specification whose arguments are any of the above
- An expression that concatenates (using CONCAT or ||) any of the above

search-string
An expression that specifies the string for which to search. *search-string* must return a value that is a built-in character string data type, graphic string data type, or binary string data type with an actual length that is no greater than 4000 bytes.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.The expression can be specified by any of the following items:
- A constant
- A special register
- A variable
- A scalar function whose arguments are any of the above (though nested function invocations cannot be used)
- An array element specification
- A CAST specification whose arguments are any of the above
- An expression that concatenates (using CONCAT or ||) any of the above

These rules are similar to those that are described for *pattern-expression* for the LIKE predicate.

A column name cannot be specified for *search-string*, except in some cases where the database manager rewrites the SQL and replaces the column name with a literal value or a variable.

The first and second arguments must have compatible string types. For more information on compatibility, see "Conversion rules for comparisons" on page 138.

If the *search-string* and *source-string* have different CCSID sets, then the *search-string* is converted to the CCSID set of the *source-string*.

Both *search-string* and *source-string* have zero or more contiguous positions. For character strings and binary strings, a position is a byte. For graphic strings, a position is a DBCS character. Graphic Unicode data is treated as UTF-16 data; a UTF-16 supplementary character takes two DBCS characters to represent and as such is counted as two DBCS characters.

The strings can contain mixed data.

- For ASCII data, if *search-string* or *source-string* contains mixed data, *search-string* is found only if the same combination of single-byte and double-byte characters are found in *source-string* in exactly the same positions.
- For EBCDIC data, if *search-string* or *source-string* contains mixed data, *search-string* is found only if any shift-in or shift-out characters are found in *source-string* in exactly the same positions, ignoring any redundant shift characters.
- For UTF-8 data, if *search-string* or *source-string* contains mixed data, *search-string* is found only if the same combination of single-byte and multi-byte characters are found in *source-string* in exactly the same position.

POSSTR operates on a strict byte-count basis without regard to single-byte or double-byte characters. It is recommended that if either the *search-string* or *source-string* contains mixed data, POSITION should be used instead of POSSTR. The POSITION function operates on a character basis. In an EBCDIC encoding scheme, any shift-in and shift-out characters are not required to be in exactly the same position and their only significance is to indicate which characters are SBCS and which characters are DBCS.

The result of the function is a large integer. The value of the result is determined by applying these rules in the order in which they appear:

- If the length of *search-string*is zero, the result is 1.
- If the length of *source-string* is zero, the result is 0.
- If the value of *search-string* is equal to an identical length substring of contiguous positions from the value of *source-string*, the result is the starting position of the first such substring within the value of *source-string*.
- If none of the above conditions are met, the result is 0.

The result can be null; if any argument is null, the result is the null value.

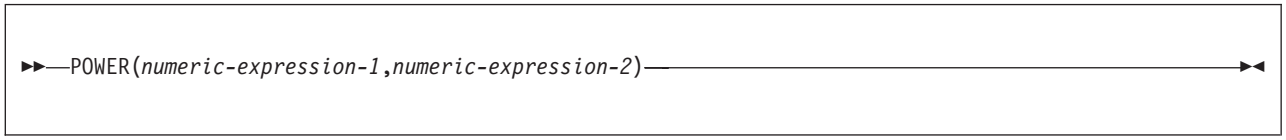
Example: Select the RECEIVED column, the SUBJECT column, and the starting position of the string 'GOOD BEER' within the NOTE_TEXT column for all rows in the IN_TRAY table that contain that string.

```
SELECT RECEIVED, SUBJECT, POSSTR(NOTE_TEXT, 'GOOD BEER')
  FROM IN_TRAY
  WHERE POSSTR(NOTE_TEXT, 'GOOD BEER') <> 0;
```

Related reference:
"LOCATE" on page 510
"LOCATE_IN_STRING" on page 513
"POSITION" on page 566

POWER

The POWER function returns the value of the first argument to the power of the second argument.



The schema is SYSIBM.

Each argument must be an expression that returns the value of any built-in numeric data type. If either argument includes a DECIMAL or REAL data type, but not a DECFLOAT data type, the arguments are converted to a double precision floating-point number for processing by the function. If either argument includes a DECFLOAT data type, the arguments are converted to DECFLOAT for processing by the function.

The result of the function depends on the data type of the arguments:
- If both arguments are SMALLINT or INTEGER, the result is INTEGER.
- If either argument is a DECFLOAT, the data type of the result is DECFLOAT(34).

  If either argument is a DECFLOAT and one of the following statements is true, the result is NaN and an invalid operation condition:
  - both arguments are zero
  - the second argument has a non-zero fractional part
  - the second argument has more than 9 digits
  - the second argument is Infinite
- Otherwise, the result is DOUBLE.

The result can be null; if any argument is null, the result is the null value.

Example 1: Assume that host variable *HPOWER* is INTEGER with a value of 3. The following statement returns the value 8.

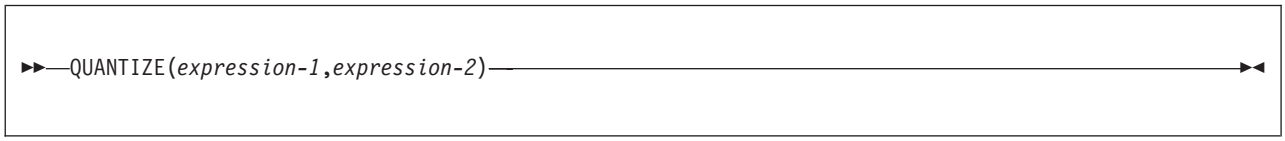
```
SELECT POWER(2,:HPOWER)
  FROM SYSIBM.SYSDUMMY1;
```

Example 2: The following statement returns the value 1.

```
SELECT POWER(0,0)
  FROM SYSIBM.SYSDUMMY1;
```

QUANTIZE

The QUANTIZE function returns a DECFLOAT value that is equal in value (except for any rounding) and sign to the first argument and that has an exponent that is set to equal the exponent of the second argument.



The schema is SYSIBM.

The number of digits that is returned (16 or 34) is the same as the number of digits in *expression-1*.

expression-1
The argument must be an expression that returns a value of any built-in numeric data type. If the argument is not a DECFLOAT value, it is converted to DECFLOAT(34) for processing.

The argument can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34).

expression-2
The argument must be an expression that returns a value of any built-in numeric data type. If the argument is not a DECFLOAT value, it is converted to DECFLOAT(34) for processing. *expression-2* is an expression that is used as an example pattern that will be used to rescale *expression-1*. The sign and coefficient of the second argument are ignored.

The argument can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34).

If one argument (after conversion) is DECFLOAT(16) and the other is DECFLOAT(34), the DECFLOAT(16) argument is converted to DECFLOAT(34) before the function is processed.

The coefficient of the result is derived from that of *expression-1*. It is rounded, if necessary (if the exponent is being increased), multiplied by a power of ten (if the exponent is being decreased), or remains unchanged (if the exponent is already equal to that of *expression-2*).

For static SQL statements other than CREATE VIEW, the ROUNDING bind option or the native SQL procedure option determines the rounding mode.

For dynamic SQL statements (and static CREATE VIEW statements), the special register CURRENT DECFLOAT ROUNDING MODE determines the rounding mode.

Unlike other arithmetic operations on the DECFLOAT data type, if the length of the coefficient after the quantize operation is greater than the precision specified by *expression-2*, a warning occurs. This ensures that, unless there is an error condition, the exponent of the result of QUANTIZE is always equal to that of *expression-2*. Furthermore:

- If either argument is NaN, NaN is returned

- If either argument is sNaN, NaN is returned and an exception occurs
- If both arguments are infinity (positive or negative), infinity (positive or negative) is returned.
- If one argument is infinity (positive or negative) and the other argument is not infinity (positive or negative), NaN is returned and an exception occurs

The result of the function is a DECFLOAT(16) value if both arguments are DECFLOAT(16). Otherwise, the result of the function is a DECFLOAT(34) value.

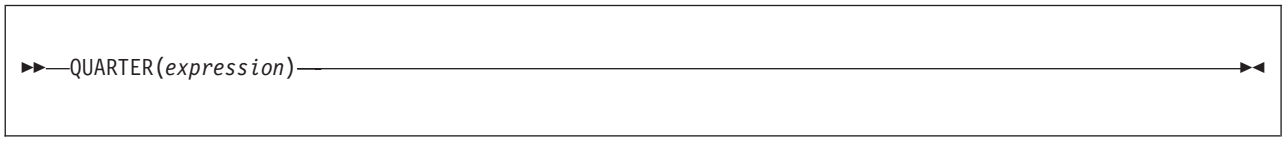
The result can be null; if any argument is null, the result is the null value.

Examples: The following examples illustrate the value that is returned for the QUANTIZE function given the input DECFLOAT values:

```
QUANTIZE(2.17, DECFLOAT(0.001))   = 2.170
QUANTIZE(2.17, DECFLOAT(0.01))    = 2.17
QUANTIZE(2.17, DECFLOAT(0.1))     = 2.2
QUANTIZE(2.17, DECFLOAT('1E+0'))  = 2
QUANTIZE(2.17, DECFLOAT('1E+1'))  = 0E+1
QUANTIZE(2, DECFLOAT(INFINITY))   = NAN   -- exception
QUANTIZE(-0.1, DECFLOAT(1) )      = 0
QUANTIZE(0, DECFLOAT('1E+5'))     = 0E+5
QUANTIZE(217, DECFLOAT('1E-1'))   = 217.0
QUANTIZE(217, DECFLOAT('1E+0'))   = 217
QUANTIZE(217, DECFLOAT('1E+1'))   = 2.2E+2
QUANTIZE(217, DECFLOAT('1E+2'))   = 2E+2
```

QUARTER

The QUARTER function returns an integer between 1 and 4 that represents the quarter of the year in which the date resides. For example, any dates in January, February, or March return the integer 1.



The schema is SYSIBM.

The argument must be an expression that returns one of the following built-in data types: a date, a timestamp, a character string, or a graphic string. If *expression* is a character or graphic string data type, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp with an actual length of not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 101.

If *expression* is a timestamp with a time zone, or a valid string representation of a timestamp with a time zone, the result is determined from the UTC representation of the datetime value.

The result of the function is a large integer.

The result can be null; if the argument is null, the result is the null value.

Example 1: The following function returns 3 because August is in the third quarter of the year.

```
SELECT QUARTER('2008-08-25')
  FROM SYSIBM.SYSDUMMY1
```

Example 2: Using sample table DSN8B10.PROJ, set the integer host variable *QUART* to the quarter of the year in which activity number 70 for project 'AD3111' occurred. Activity completion dates are recorded in column ACENDATE.

```
SELECT QUARTER(ACENDATE)
  INTO :QUART
  FROM DSN8B10.PROJ
  WHERE PROJNO = 'AD3111' AND ACTNO = 70;
```

QUART is set to 4.

Example 3: The following invocations of the QUARTER function returns the same result:

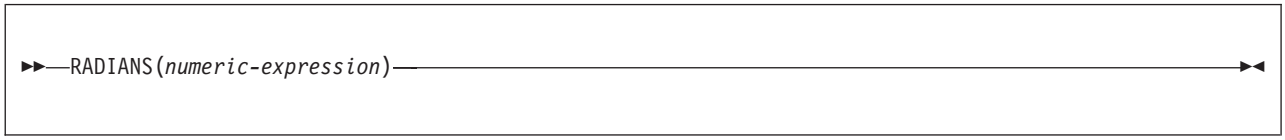
```
SELECT QUARTER('2003-01-02-20.10.05.123456'),
   QUARTER('2003-01-02-12.10.05.123456-08:00'),
   QUARTER('2003-01-03-05.10.05.123456+09:00')
  FROM SYSIBM.SYSDUMMY1;
```

For each invocation of the QUARTER function in this SELECT statement, the result is 1.

When the input argument contains a time zone, the result is determined from the UTC representation of the input value. The string representations of a timestamp with a time zone in the SELECT statement all have the same UTC representation: 2003-01-02-20.10.05.123456. The month portion of the UTC representation is 1 for January, which is in the first quarter.

RADIANS

The RADIANS function returns the number of radians for an argument that is expressed in degrees.



The schema is SYSIBM.

The argument must be an expression that returns the value of any built-in numeric data type that is not DECFLOAT. If the argument is not a double precision floating-point number, it is converted to one for processing by the function.

The result of the function is a double precision floating-point number.

The result can be null; if the argument is null, the result is the null value.

Example: Assume that host variable *HDEG* is an INTEGER with a value of 180. The following statement returns a double precision floating-point number with an approximate value of 3.1415926536.

```
SELECT RADIANS(:HDEG)
  FROM SYSIBM.SYSDUMMY1;
```

RAISE_ERROR

The RAISE_ERROR function causes the statement that invokes the function to return an error with the specified SQLSTATE (along with SQLCODE -438) and error condition. The RAISE_ERROR function always returns the null value with an undefined data type.

The schema is SYSIBM.

sqlstate
An expression that returns a character string (CHAR or VARCHAR) of exactly 5 characters.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR(5) data type.The *sqlstate* value must follow these rules for application-defined SQLSTATEs:

- Each character must be from the set of digits ('0' through '9') or non-accented upper case letters ('A' through 'Z').
- The SQLSTATE class (first two characters) cannot be '00', '01', or '02' because these are not error classes.
- If the SQLSTATE class (first two characters) starts with the character '0' through '6' or 'A' through 'H', the subclass (last three characters) must start with a letter in the range 'I' through 'Z'.
- If the SQLSTATE class (first two characters) starts with the character '7', '8', '9', or 'I' though 'Z', the subclass (last three characters) can be any of '0' through '9' or 'A through 'Z'.

diagnostic-string
An expression that returns a character string with a data type of CHAR or VARCHAR and a length of up to 70 bytes. The string contains EBCDIC data that describes the error condition. If the string is longer than 70 bytes, it is truncated.

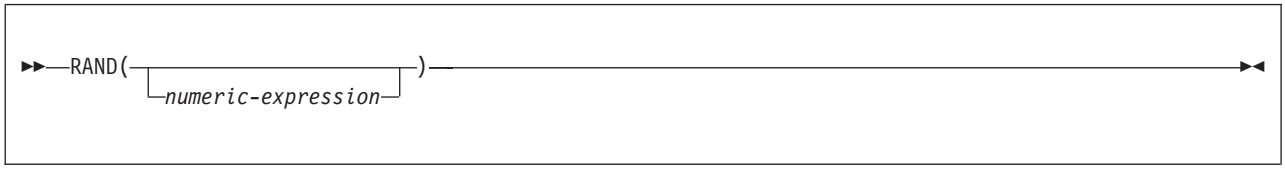
Since the data type of the result of RAISE_ERROR is undefined, it can only be used in a SET *host-variable* or SQL procedure language*assignment-statement*. To use this function in another context, such as alone in a select list, you must use a cast specification to give a data type to the null value that is returned. The RAISE_ERROR function is most useful with CASE expressions.

Example: For each employee in sample table DSN8B10.EMP, list the employee number and education level. List the education level as 'Post Graduate', 'Graduate' and 'Diploma' instead of the integer that it is stored as in the table. If an education level is greater than '20', raise an error ('70001') with a description.

```
SELECT EMPNO,
  CASE WHEN EDLEVEL < 16 THEN 'Diploma'
       WHEN EDLEVEL < 18 THEN 'Graduate'
       WHEN EDLEVEL < 21 THEN 'Post Graduate'
       ELSE RAISE_ERROR('70001',
                        'EDUCLVL has a value greater than 20')
  END
  FROM DSN8B10.EMP;
```

RAND

The RAND function returns a random floating-point value between 0 and 1. An argument can be specified as an optional seed value.



The schema is SYSIBM.

numeric-expression

If *numeric-expression* is specified, it is used as the seed value. The argument must be an expression that returns a value of a built-in integer data type (SMALLINT or INTEGER). The value must be between 0 and 2,147,483,646.

The argument can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34) and then assigned to an INTEGER value.

The result of the function is a double precision floating-point number.

The result can be null; if the argument is null, the result is the null value.

A specific seed value, other than zero, will produce the same sequence of random numbers for a specific instance of a RAND function in a query each time the query is executed. The seed value is used only for the first invocation of an instance of the RAND function within a statement. `RAND(0)` is processed the same as `RAND()`.

The RAND function is a not deterministic.

Example: Assume that host variable *HRAND* is an INTEGER with a value of 100. The following statement returns a random floating-point number between 0 and 1, such as the approximate value .0121398:

```
SELECT RAND(:HRAND)
  FROM SYSIBM.SYSDUMMY1;
```

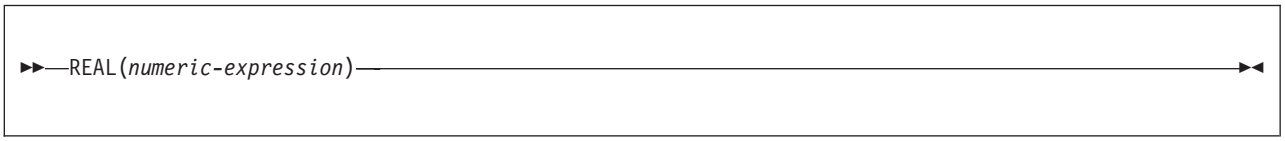
To generate values in a numeric interval other than 0 to 1, multiply the RAND function by the size of the interval that you want. For example, to get a random number between 0 and 10, such as the approximate value 5.8731398, multiply the function by 10:

```
SELECT (RAND(:HRAND) * 10)
  FROM SYSIBM.SYSDUMMY1;
```

REAL

The REAL function returns a single-precision floating-point representation of either a number or a string representation of a number.

Numeric to Real:



String to Real:

The schema is SYSIBM.

Numeric to Real

numeric-expression
An expression that returns a value of any built-in numeric data type.

The result is the same number that would occur if the argument were assigned to a single precision floating-point column or variable. If the numeric value of the argument is not within the range of single precision floating-point, an error occurs.

String to Real

string-expression
An expression that returns a value of a character or graphic string (except a CLOB or DBCLOB) with a length attribute that is not greater than 255 bytes. The string must contain a valid string representation of a number.

The result is the same number that would result from `CAST(`*string-expression*` AS REAL)`. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an SQL floating-point, integer, or decimal constant.

The result of the function is a single precision floating-point number.

The result can be null; if the argument is null, the result is the null value.

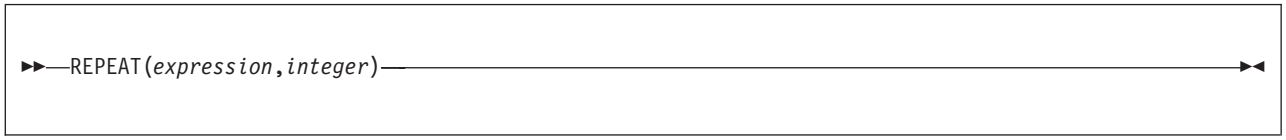
Recommendation: To increase the portability of applications, use the CAST specification. For more information, see "CAST specification" on page 267.

Example: Using sample table DSN8B10.EMP, find the ratio of salary to commission for employees whose commission is not zero. The columns involved, SALARY and COMM, have decimal data types. To express the result in single precision floating-point, apply REAL to SALARY so that the division is carried out in floating-point (actually double precision) and then apply REAL to the complete expression so that the results are returned in single precision floating-point.

```
SELECT EMPNO, REAL(REAL(SALARY)/COMM)
  FROM DSN8B10.EMP
  WHERE COMM > 0;
```

REPEAT

The REPEAT function returns a character string that is composed of an argument that is repeated a specified number of times.



The schema is SYSIBM.

expression
An expression that specifies the string to be repeated. The expression must return a value that is a built-in character string, graphic string, or binary string data type that is not a LOB.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.The actual length of the string must be greater or equal to 1 and less than or equal to 32704 bytes.

integer
integer must be a positive large integer value that specifies the number of times to repeat the string.

The argument can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34) which is then assigned to an INTEGER value.

The result can be null; if any argument is null, the result is the null value.

The encoding scheme of the result is the same as *expression*. The data type of the result of the function depends on the data type of *expression*:
- VARBINARY if *expression* is a binary string
- VARCHAR if *expression* is a character string
- VARGRAPHIC if *expression* is graphic string

The CCSID of the result is the same as the CCSID of *expression*.

If *integer* is a constant, the length attribute of the result is the length attribute of *expression* times *integer*. Otherwise, the length attribute depends on the data type of the result:
- 4000 for VARBINARY and VARCHAR
- 2000 for VARGRAPHIC

The actual length of the result is the actual length of *expression* times *integer*. If the actual length of the result string exceeds the maximum for the return type, an error occurs.

Example 1: Repeat 'abc' two times to create 'abcabc'.

```
SELECT REPEAT('abc',2)
  FROM SYSIBM.SYSDUMMY1;
```

Example 2: List the phrase 'REPEAT THIS' five times. Use the CHAR function to limit the output to 60 bytes.

```
SELECT CHAR(REPEAT('REPEAT THIS',5), 60)
  FROM SYSIBM.SYSDUMMY1;
```

This example results in 'REPEAT THISREPEAT THISREPEAT THISREPEAT THISREPEAT THIS '.

Example 3: For the following query, the LENGTH function returns a value of 0 because the result of repeating a string zero times is an empty string, which is a zero-length string.

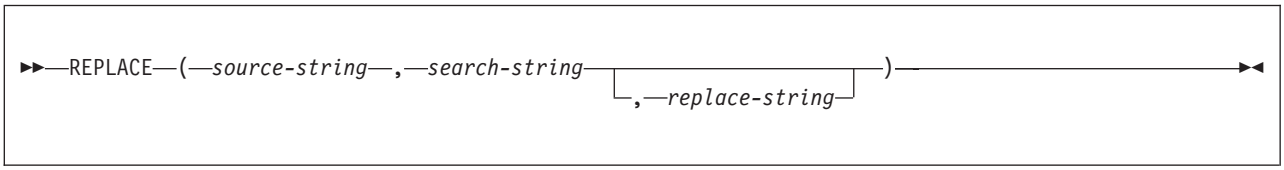
```
SELECT LENGTH(REPEAT('REPEAT THIS',0))
  FROM SYSIBM.SYSDUMMY1;
```

Example 4: For the following query, the LENGTH function returns a value of 0 because the result of repeating an empty string any number of times is an empty string, which is a zero-length string.

```
SELECT LENGTH(REPEAT('', 5))
  FROM SYSIBM.SYSDUMMY1;
```

REPLACE

The REPLACE function replaces all occurrences of *search-string* in *source-string* with *replace-string*. If *search-string* is not found in *source-string*, *source-string* is returned unchanged.



The schema is SYSIBM.

source-string
An expression that specifies the source string. The expression must return a value that is a built-in character string, graphic string, or binary string data type that is not a LOB and it cannot be an empty string.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.The length of *source-string* must be greater than or equal to the length of *search-string*.

search-string
An expression that specifies the string to be removed from the source string. The expression must return a value that is a built-in character string, graphic string, or binary string data type that is not a LOB; the value cannot be an empty string.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.

replace-string
An expression that specifies the replacement string. The expression must return a value that is a built-in character string, graphic string, or binary string data type that is not a LOB.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.If *replace-string* is not specified or is an empty string, nothing replaces the string that is removed from the source string.

The actual length of each string must be32764 bytes or less for character and binary strings or 16382 or less for graphic strings.

All three arguments must have compatible data types. If the expressions have different CCSID sets, then the expressions are converted to the CCSID set of *source-string*.

The data type of the result of the function depends on the data type of *source-string*, *search-string*, and *replace-string*:

- VARCHAR if *source-string* is a character string. The encoding scheme of the result is the same as *source-string*. The CCSID of the result depends on the arguments:
  - If *source-string*, *search-string*, or *replace-string* is bit data, the result is bit data.
  - If *source-string*, *search-string*, and *replace-string* are all SBCS Unicode data, the CCSID of the result is the CCSID for SBCS Unicode data.

  - If *source-string* is SBCS Unicode data, and *search-string* or *replace-string* is not SBCS Unicode data, the CCSID of the result is the mixed CCSID for Unicode data.
  - Otherwise, the CCSID of the result is the mixed CCSID that corresponds to the CCSID of *source-string*. However, if the input is EBCDIC or ASCII and there is no corresponding system CCSID for mixed, the CCSID of the result is the CCSID of *source-string*.
- VARGRAPHIC if *source-string* is a graphic. The encoding scheme of the result is the same as *source-string*. The CCSID of the result is the same as the CCSID of *source-string*.
- VARBINARY if *source-string*, *search-string*, and *replace-string* are binary strings.

The length attribute of the result depends on the arguments:

- If the length attribute of *replace-string* is less than or equal to the length attribute of *search-string*, the length attribute of the result is the length attribute of *source-string*.
- If the length attribute of *replace-string* is greater than the length attribute of *search-string*, the length attribute of the result is determined as follows depending on the data type of the result:
  - For VARCHAR or VARBINARY:
    - If *L1* < = 4000, the length attribute of the result is `MIN(4000, (L3*(L1/L2)) + MOD(L1,L2))`
    - Otherwise, the length attribute of the result is `MIN(32764, (L3*(L1/L2)) + MOD(L1,L2))`
  - For VARGRAPHIC:
    - If *L1* < = 2000, the length attribute of the result is `MIN(2000, (L3*(L1/L2)) + MOD(L1,L2))`
    - Otherwise, the length attribute of the result is `MIN(16382, (L3*(L1/L2)) + MOD(L1,L2))`

  where:
  L1 is the length attribute of *source-string*
  L2 is the length attribute of *search-string* if the search string is a string constant. Otherwise, L2 is 1.
  L3 is the length attribute of *replace-string*

If the result is a character string or binary string, the length attribute of the result must not exceed 32764. If the result is a graphic string, the length attribute of the result must not exceed 16382.

The actual length of the result is the actual length of *source-string* plus the number of occurrences of *search-string* that exist in *source-string* multiplied by the actual length of *replace-string* minus the actual length of *search-string*. If the actual length of the result string exceeds the maximum for the return data type, an error occurs.

The result can be null; if any argument is null, the result is the null value.

Example 1: Replace all occurrences of the character 'N' in the string 'DINING' with 'VID'. Use the CHAR function to limit the output to 10 bytes.

```
SELECT CHAR(REPLACE('DINING','N','VID'),10)
  FROM SYSIBM.SYSDUMMY1;
```

The result is the string 'DIVIDIVIDG'.

Example 2: Replace string 'ABC' in the string 'ABCXYZ' with nothing, which is the same as removing 'ABC' from the string.

```
SELECT REPLACE('ABCXYZ','ABC','')
  FROM SYSIBM.SYSDUMMY1;
```

The result is the string 'XYZ'.

Example 3: Replace string 'ABC' in the string 'ABCCABCC' with 'AB'. This example illustrates that the result can still contain the string that is to be replaced (in this case, 'ABC') because all occurrences of the string to be replaced are identified prior to any replacement.

```
SELECT REPLACE('ABCCABCC','ABC','AB')
  FROM SYSIBM.SYSDUMMY1;
```

The result is the string 'ABCABC'.

Related concepts:

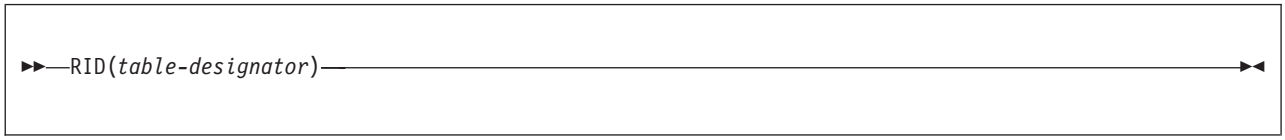
"Character strings" on page 84

"Binary strings" on page 95

"Graphic strings" on page 94

RID

The RID function returns the record ID (RID) of a row. The RID is used to uniquely identify a row.



The schema is SYSIBM.

The function might return a different value when it is invoked multiple times for a row. For example, after the REORG utility is run, the RID function might return a different value for a row than would have been returned prior to the REORG utility being run. The RID function is not deterministic.

table-designator
table-designator must uniquely identify a base table, a view, or a nested table expression of a subselect in which the function is referenced.

If *table-designator* specifies a view or a nested table expression, the RID function returns the RID of the base table of the view or nested table expression. The specified view or nested table expression must contain only one base table in its outer subselect. *table-designator* must not specify a view or a nested table expression that is materialized.

table-designator must not specify a table function or a collection-derived table.

The result of the function is BIGINT. The result can be null.

Considerations for RID values: DB2 might reuse RID numbers when a REORG operation is performed. If the RID function is used to obtain a value for a row and an application depends on that value remaining the same as long as the row exists, consider the following alternatives:

- Add a ROWID column to the table to provide a value that can be associated with each row, rather than invoking the RID function to generate a value for a row.
- Define a primary key for the table, using the columns of the primary key to ensure uniqueness, rather than invoking the RID function to generate a value for a row.

Example 1: Return the RID and last name of employees who are in department '20':

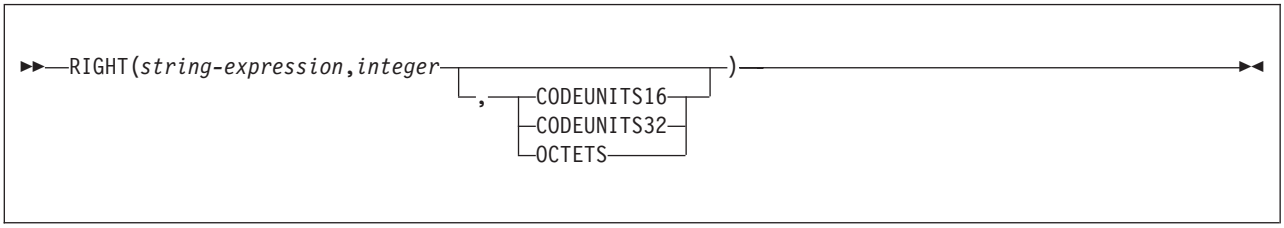
```
SELECT RID(EMP), LASTNAME
  FROM EMP
  WHERE DEPTNO = '20';
```

Example 2: Set the host variable *HV_EMP_RID* as the value of the RID for the employee with the employee number of '3500':

```
SELECT RID(EMP) INTO :HV_EMP_RID
  FROM EMP
  WHERE EMPNO = '3500';
```

RIGHT

The RIGHT function returns a string that consists of the specified number of rightmost bytes or specified string unit from a string.



The schema is SYSIBM.

string-expression
An expression that specifies the string from which the result is derived. The string must be any built-in string data type. A substring of *string-expression* is zero or more contiguous code points of *string-expression*. A partial surrogate character in the expression is replaced with a blank.

The string can contain mixed data. Depending on the units that are specified to evaluate the function, the result is not necessarily a properly formed mixed data character string.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.

integer
An expression that specifies the length of the result. The value must be an integer between 0 and *n*, where *n* is the length attribute of *string-expression*, expressed in the units that are either implicitly or explicitly specified.

The argument can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34) which is then assigned to an INTEGER value.

If CODEUNITS16 or CODEUNITS32 is specified, see "Determining the length attribute of the final result" on page 90 for information about how to calculate the length attribute of the result string.

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the unit that is used to express *integer*. If *string-expression* is a character string that is defined as bit data, CODEUNITS16 and CODEUNITS32 cannot be specified. If *string-expression* is a graphic string, OCTETS cannot be specified. If *string-expression* is a binary string, CODEUNITS16, CODEUNITS32, and OCTETS cannot be specified.

CODEUNITS16
Specifies that *integer* is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32
Specifies that *integer* is expressed in terms of 32-bit UTF-32 code units.

OCTETS
Specifies that *integer* is expressed in terms of bytes.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String unit specifications" on page 87.

The *string-expression* is effectively padded on the right with the necessary number of padding characters so that the specified substring of *string-expression* always exists. The encoding scheme of the data determines the padding character:

- For ASCII SBCS data or ASCII mixed data, the padding character is X'20'.
- For ASCII DBCS data, the padding character depends on the CCSID; for example, for Japanese (CCSID 301) the padding character is X'8140', while for simplified Chinese it is X'A1A1'.
- For EBCDIC SBCS data or EBCDIC mixed data, the padding character is X'40'.
- For EBCDIC DBCS data, the padding character is X'4040'.
- For Unicode SBCS data or UTF-8 data (Unicode mixed data), the padding character is X'20'.
- For UTF-16 data (Unicode DBCS data), the padding character is X'0020'.
- For binary data, the padding character is X'00'.

The result of the function is a varying-length string with a length attribute that is the same as the length attribute of *string-expression* and a data type that depends on the data type of *string-expression*:

- VARBINARY if *string-expression* is BINARY or VARBINARY
- VARCHAR if *string-expression* is CHAR or VARCHAR
- CLOB if *string-expression* is CLOB
- VARGRAPHIC if *string-expression* is GRAPHIC or VARGRAPHIC
- DBCLOB if *string-expression* is DBCLOB
- BLOB if *string-expression* is BLOB

The actual length of the result is determined from *integer*.

The result can be null; if any argument is null, the result is the null value.

The CCSID of the result is the same as that of *string-expression*.

Example 1: Assume that host variable *ALPHA* has a value of 'ABCDEF'. The following statement returns the value 'DEF', which are the three rightmost characters in *ALPHA*.

```
SELECT RIGHT(ALPHA,3)
  FROM SYSIBM.SYSDUMMY1;
```

Example 2: The following statement returns a zero length string.

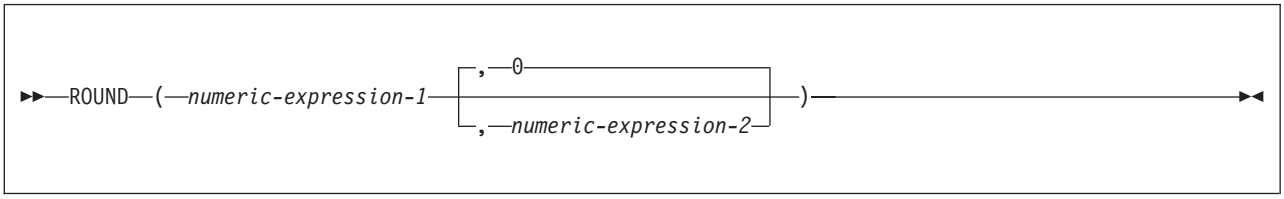
```
SELECT RIGHT('ABCABC',0)
  FROM SYSIBM.SYSDUMMY1;
```

Example 3: FIRSTNME is a VARCHAR(12) column in table T1. When FIRSTNME has the 6-character string 'Jürgen' as a value:

```
Function ...                        Returns ...
RIGHT(FIRSTNME,5,CODEUNITS32)       'ürgen'  -- x'C3BC7267656E'
RIGHT(FIRSTNME,5,CODEUNITS16)       'ürgen'  -- x'C3BC7267656E'
RIGHT(FIRSTNME,5,OCTETS)            ' rgen'  -- x'207267656E' a truncated string
```

ROUND

The ROUND function returns a number that is rounded to the specified number of places to the right or left of the decimal place.



The schema is SYSIBM.

numeric-expression-1
An expression that returns a value of any built-in numeric data type.

If *expression-1* is a decimal floating-point data type, the DECFLOAT ROUNDING MODE will not be used. The rounding behavior of ROUND corresponds to a value of ROUND_HALF_UP. If you want a different rounding behavior, use the QUANTIZE function.

The argument can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34).

numeric-expression-2
An expression that returns a value of a built-in small integer data type or large integer data type.

The absolute value of integer specifies the number of places to the right of the decimal point for the result if *numeric-expression-2* is not negative. If *numeric-expression-2* is negative, *numeric-expression-1* is rounded to the sum of the absolute value of *numeric-expression-2*+1 number of places to the left of the decimal point.

If the absolute value of *numeric-expression-2* is larger than the number of digits to the left of the decimal point, the result is 0. (For example, `ROUND(748.58,-4)` returns 0.)

If *numeric-expression-1* is positive, a digit value of 5 is rounded to the next higher positive number. If *numeric-expression-1* is negative, a digit value of 5 is rounded to the next lower negative number.

The argument can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34).

The result of the function has the same data type and length attribute as the first argument except that the precision is increased by one if the argument is DECIMAL and the precision is less than 31. For example, an argument with a data type of DECIMAL(5,2) results in DECIMAL(6,2). An argument with a data type of DECIMAL(31,2) results in DECIMAL(31,2).

The result can be null; if any argument is null, the result is the null value.

Example 1: Calculate the number '873.726' rounded to '2', '1', '0', '-1', and '-2' decimal places respectively.

```
SELECT ROUND(873.726,2),
       ROUND(873.726,1),
       ROUND(873.726,0),
```

```
       ROUND(873.726,-1),
       ROUND(873.726,-2),
       ROUND(873.726,-3),
       ROUND(873.726,-4)
  FROM SYSIBM.SYSDUMMY1;
```

This example returns the values '0873.730', '0873.700', '0874.000', '0870.000', '0900.000', '1000.000', and '0000.000'.

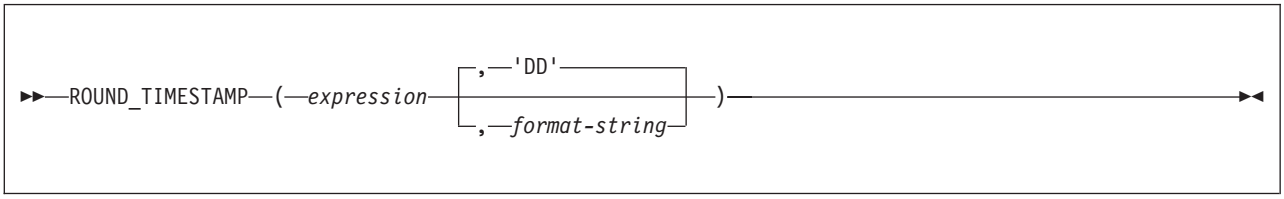
Example 2: To demonstrate how numbers are rounded in positive and negative values, calculate the numbers '3.5', '3.1', '-3.1', '-3.5' rounded to '0' decimal places.

```
SELECT ROUND(3.5,0),
       ROUND(3.1,0),
       ROUND(-3.1,0),
       ROUND(-3.5,0)
  FROM SYSIBM.SYSDUMMY1;
```

This example returns the values '04.0', '03.0', '-03.0', and '-04.0'. (Notice that in the positive value '3.5' is rounded up to the next higher number while in the negative value '-3.5' is rounded down to the next lower negative number.)

ROUND_TIMESTAMP

The ROUND_TIMESTAMP scalar function returns a timestamp that is rounded to the unit that is specified by the timestamp format string.



The schema is SYSIBM.

expression
An expression that returns a value of any of the following built-in data types: a timestamp, a character string, or a graphic string. If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a timestamp with an actual length that is not greater than 255 bytes. A time zone in a string representation of a timestamp is ignored. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 101.

format-string
An expression that returns a built-in character string or graphic string data type, with a length that is not greater than 255 bytes. *format-string* contains a template of how the timestamp represented by expression should be rounded. For example, if *format-string* is 'DD', the timestamp that is represented by *expression* is rounded to the nearest day. *format-string* must be a valid template for a timestamp, and not include leading or trailing blanks.

Allowable values for *format-string* are listed in the following table.

Table 68. ROUND_TIMESTAMP and TRUNC_TIMESTAMP format models

Format model	Rounding or truncating unit	ROUND_TIMESTAMP example	TRUNC_TIMESTAMP example
CC SCC	Century Rounds up to the start of the next century after the 50th year of the century (for example on 1951–01–01–00.00.00). Not valid for a TIME argument.	Input Value: 1897-12-04-12.22.22.000000 Result: 1901-01-01-00.00.00.000000	Input Value: 1897-12-04-12.22.22.000000 Result: 1801-01-01-00.00.00.000000
SYYYY YYYY YEAR SYEAR YYY YY Y	Year (Rounds up on July 1st)	Input Value: 1897-12-04-12.22.22.000000 Result: 1898-01-01-00.00.00.000000	Input Value: 1897-12-04-12.22.22.000000 Result: 1897-01-01-00.00.00.000000
IYYY IYY IY I	ISO Year (Rounds up on July 1st)	Input Value: 1897-12-04-12.22.22.000000 Result: 1898-01-03-00.00.00.000000	Input Value: 1897-12-04-12.22.22.000000 Result: 1897-01-04-00.00.00.000000

Table 68. ROUND_TIMESTAMP and TRUNC_TIMESTAMP format models (continued)

Format model	Rounding or truncating unit	ROUND_TIMESTAMP example	TRUNC_TIMESTAMP example
Q	Quarter (Rounds up on the sixteenth day of the second month of the quarter)	Input Value: 1999-06-04-12.12.30.000000 Result: 1999-07-01-00.00.00.000000	Input Value: 1999-06-04-12.12.30.000000 Result: 1999-04-01-00.00.00.000000
MONTH MON MM RM	Month (Rounds up on the sixteenth day of the month)	Input Value: 1999-06-18-12.12.30.000000 Result: 1999-07-01-00.00.00.000000	Input Value: 1999-06-18-12.15.00.000000 Result: 1999-06-01-00.00.00.000000
WW	Same day of the week as the first day of the year (Rounds up on the 12th hour of the 3rd day of the week, with respect to the first day of the year)	Input Value: 2000-05-05-12.12.30.000000 Result: 2000-05-06-00.00.00.000000	Input Value: 2000-05-05-12.15.00.000000 Result: 2000-04-29-00.00.00.000000
IW	Same day of the week as the first day of the ISO year (Rounds up on the 12th hour of the 3rd day of the week, with respect to the first day of the ISO year)	Input Value: 2000-05-05-12.12.30.000000 Result: 2000-05-08-00.00.00.000000	Input Value: 2000-05-05-12.15.00.000000 Result: 2000-05-01-00.00.00.000000
W	Same day of the week as the first day of the month (Rounds up on the 12th hour of the 3rd day of the week, with respect to the first day of the month)	Input Value: 2000-05-17-12.12.30.000000 Result: 2000-05-15-00.00.00.000000	Input Value: 2000-05-17-12.15.00.000000 Result: 2000-05-15-00.00.00.000000
DDD DD J	Day (Rounds up on the 12th hour of the day)	Input Value: 2000-05-17-12.59.59.000000 Result: 2000-05-18-00.00.00.000000	Input Value: 2000-05-17-12.59.59.000000 Result: 2000-05-17-00.00.00.000000
DAY DY D	Starting day of the week (Rounds up with respect to the 12th hour of the third day of the week. The first day of the week is always Sunday).	Input Value: 2000-05-17-12.59.59.000000 Result: 2000-05-21-00.00.00.000000	Input Value: 2000-05-17-12.59.59.000000 Result: 2000-05-14-00.00.00.000000
HH HH12 HH24	Hour (Rounds up at 30 minutes)	Input Value: 2000-05-17-23.59.59.000000 Result: 2000-05-18-00.00.00.000000	Input Value: 2000-05-17-23.59.59.000000 Result: 2000-05-17-23.00.00.000000
MI	Minute (Rounds up at 30 seconds)	Input Value: 2000-05-17-23.58.45.000000 Result: 2000-05-17-23.59.00.000000	Input Value: 2000-05-17-23.58.45.000000 Result: 2000-05-17-23.58.00.000000
SS	Second (Rounds up at 500000 microseconds)	Input Value: 2000-05-17-23.58.45.500000 Result: 2000-05-17-23.58.46.000000	Input Value: 2000-05-17-23.58.45.500000 Result: 2000-05-17-23.58.45.000000

If *expression* does not have data type TIMESTAMP WITHOUT TIME ZONE, *expression* is cast as follows:

- If *expression* is a TIMESTAMP WITH TIME ZONE value, *expression* is cast to TIMESTAMP WITHOUT TIME ZONE, with the same precision as *expression*.
- Otherwise, *expression* is cast to TIMESTAMP(6) WITHOUT TIME ZONE.

The result of the function has the same data type as the data type to which *expression* is cast.

The result can be null; if any argument is null, the result is the null value.

The result CCSID is the appropriate CCSID of the argument encoding scheme and the result subtype is the appropriate subtype of the CCSID.

Example 1: Set the host variable *RND_TMSTMP* with the input timestamp rounded to the nearest year value.

```
SET :RND_TMSTMP = ROUND_TIMESTAMP(TIMESTAMP_FORMAT('2000-08-14 17:30:00',
                                                   'YYYY-MM-DD HH24:MI:SS'), 'YEAR');
```

The value set is '2001-01-01-00.00.00.000000'.

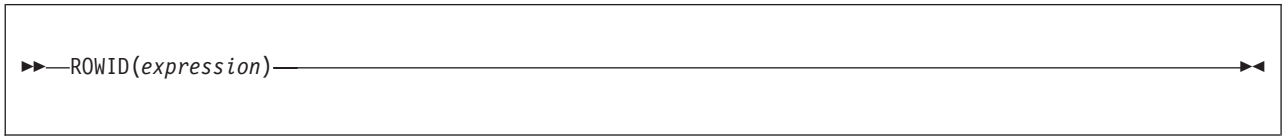
Example 2: Assume *PRSTSZ* is an SQL variable with the TIMESTAMP WITH TIME ZONE value '2008-04-15.20.00.000000-08:30'. The input value is first cast to TIMESTAMP WITHOUT TIME ZONE (as '2008-04-15.20.00.000000') for the ROUND_TIMESTAMP function.

```
SELECT ROUND_TIMESTAMP(PRSTSZ)
 FROM PROJECT;
```

The ROUND_TIMESTAMP function returns a TIMESTAMP WITHOUT TIME ZONE value of '2008-04-16.00.00.000000'.

ROWID

The ROWID function returns a row ID representation of its argument.



The schema is SYSIBM.

The argument must be an expression that returns a value of a built-in character string data type, other than a CLOB, with a maximum length that is no greater than 255 bytes. Although the character string can contain any value, it is recommended that the character string contain a ROWID value that was previously generated by DB2 to ensure a valid ROWID value is returned. For example, the function can be used to convert a ROWID value that was cast to a CHAR value back to a ROWID value.

If the actual length of *expression* is less than 40, the result is not padded. If the actual length of *expression* is greater than 40, the result is truncated. If non-blank characters are truncated, a warning is returned.

The result of the function is a ROWID value.

The length attribute of the result is 40. The actual length of the result is the length of *expression*.

The result can be null; if the argument is null, the result is the null value.

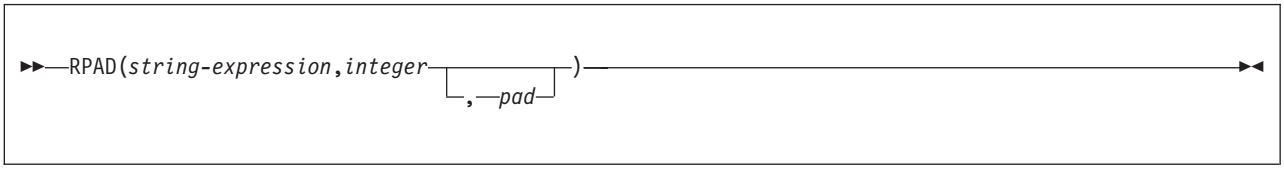
A null ROWID value cannot be used as the value for a row ID column in the database.

Example: Assume that table EMPLOYEE contains a row ID column, 'EMP_ROWID'. Also assume that the table contains a row that is identified by a ROWID value that is equivalent to X'F0DFD230E3C0D80D81C201AA0A280100000000000203'. Using direct row access, select the employee number for that row.

```
SELECT EMPNO
  FROM EMPLOYEE
  WHERE EMP_ROWID=ROWID(X'F0DFD230E3C0D80D81C201AA0A280100000000000203');
```

RPAD

The RPAD function returns a string that is padded on the right with blanks or a specified string.



The schema is SYSIBM.

The RPAD function returns a string composed of *string-expression* padded on the right, with *pad* or blanks. The RPAD function treats leading or trailing blanks in *string-expression* as significant. Padding will only occur if the actual length of *string-expression* is less than *integer*, and *pad* is not an empty string.

string-expression
An expression that specifies the source string. The expression must return a value that is a built-in string data type that is not a LOB.

integer
An integer constant that specifies the length of the result. The value must be zero or a positive integer that is less than or equal to *n*, where *n* is 32704 if *string-expression* is a character or binary string, or where *n* is 16352 if *string-expression* is a graphic string.

pad
An expression that specifies the string with which to pad. The expression must return a value that is a built-in string data type that is not a LOB. If *pad* is not specified, the pad character is determined as follows:

- SBCS blank character if *string-expression* is a character string.
- DBCS blank character if *string-expression* is a graphic string.
- Hexadecimal zero (X'00'), if *string-expression* is a binary string.

The result of the function is a varying length string that has the same CCSID of *string-expression*. *string-expression* and *pad* must have compatible data types. If the string expressions have different CCSID sets, then *pad* is converted to the CCSID set of *string-expression*. If either *string-expression* or *pad* is FOR BIT DATA, no character conversion occurs. The actual length of the result is determined from *integer*.

The length attribute of the result depends on *integer*. If *integer* is greater than 0, the length attribute of the result is *integer*. If *integer* is 0, the length attribute of the result is 1.

The actual length of the result is determined from *integer*. If *integer* is 0, the actual length is 0, and the result is the empty string. If *integer* is less than the actual length of *string-expression*, the actual length is *integer* and the result is truncated.

The result can be null; if any argument is null, the result is the null value.

Example 1: Assume that NAME is a VARCHAR(15) column that contains the values 'Chris', 'Meg', and 'Jeff'. The following query will completely pad out a value on the right with periods.

```
SELECT RPAD(NAME,15,'.' ) AS NAME
  FROM T1;
```

The results are similar to the following output:

```
NAME
---------------

Chris..........
Meg............
Jeff...........
```

Example 2: Similar to Example 1, the following query will completely pad out a value on the right with *pad* (note that in some cases there is a partial instance of the padding specification):

```
SELECT RPAD(NAME,15,'123' ) AS NAME
  FROM T1;
```

The results are similar to the following output:

```
NAME
---------------

Chris1231231231
Meg123123123123
Jeff12312312312
```

Example 3: Similarly, the following query will only pad each value to a length of 5:

```
SELECT RPAD(NAME,5,'.') AS NAME
  FROM T1;
```

The results are similar to the following output:

```
NAME
---------------

Chris
Meg..
Jeff.
```

Example 4: Assume that NAME is a CHAR(15) column that contains the values 'Chris', 'Meg', and 'Jeff'. Note that the result of RTRIM in the following example is a varying length string with the blanks removed:

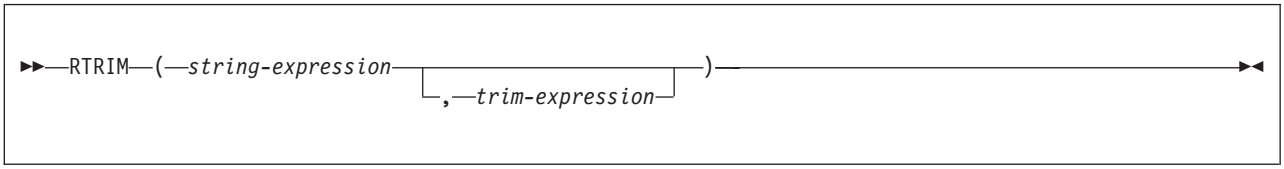
```
SELECT RPAD(RTRIM(NAME),15,'.') AS NAME
  FROM T1;
```

The results are similar to the following output:

```
NAME
---------------
Chris..........
Meg............
Jeff...........
```

RTRIM

The RTRIM function removes bytes from the end of a string expression based on the content of a trim expression.



The schema is SYSIBM.

The RTRIM function removes all of the characters contained in *trim-expression* from the end of *string-expression*. The search is done by comparing the binary representation of each character (which consists of one or more bytes) in *trim-expression* to the bytes at the end of *string-expression*. If *string-expression* is defined as FOR BIT DATA, the search is done by comparing each byte in *trim-expression* to the byte at the end of *string-expression*.

string-expression
An expression that specifies the source string. The argument must be an expression that returns a value that is a built-in string data type that is not a LOB, or a numeric data type. If the value is not a string data type, it is implicitly cast to VARCHAR before the function is evaluated. If *string-expression* is not FOR BIT DATA, *trim-expression* must not be FOR BIT DATA.

trim-expression
An expression that specifies the characters to remove from the end of *string-expression*. The expression must return a value that is a built-in string data type that is not a LOB, or a numeric data type. If the value is not a string data type, it is implicitly cast to VARCHAR before the function is evaluated.

The default for *trim-expression* depends on the data type of *string-expression*:

- A DBCS blank if *string-expression* is a DBCS graphic string. For ASCII, the CCSID determines the hex value that represents a DBCS blank. For example, for Japanese (CCSID 301), X'8140' represents a DBCS blank, while for Simplified Chinese, X'A1A1' represents a DBCS blank. For EBCDIC, X'4040' represents a DBCS blank.
- A UTF-16 or UCS-2 blank (X'0020') if *string-expression* is a Unicode graphic string.
- A value of X'00' if *string-expression* is a binary string.
- Otherwise, a single byte blank. For EBCDIC, X'40' represents a blank. When not EBCDIC, X'20' represents a blank.

string-expression and *trim-expression* must have compatible data types. If *string-expression* and *trim-expression* have different CCSID sets, *trim-expression* is converted to the CCSID of *string-expression*.

The result of the function depends on the data type of *string-expression*.

- VARCHAR if *string-expression* is a character string. If *string-expression* is defined as FOR BIT DATA, the result is FOR BIT DATA.
- VARGRAPHIC if *string-expression* is a graphic string.

- VARBINARY if *string-expression* is a binary string.

The length attribute of the result is the same as the length attribute of *string-expression*.

The actual length of the result for a character or binary string is the length of *string-expression* minus the number of bytes removed. The actual length of the result for a graphic string is the length (in number of double byte characters) of *string-expression* minus the number of double byte characters removed. If all of the characters are removed, the result is an empty string (the length is zero).

The result can be null; if the argument is null, the result is the null value.

The CCSID of the result is the same as that of *string-expression*.

Example: **Use the RTRIM function to remove individual numbers in the second argument from the end (right side) of the first argument:**

```
SELECT RTRIM ('123DEFG123', '321'),
    RTRIM ('12322XYZ12322222', '123'),
    RTRIM ('12321', '213'),
    RTRIM ('123XYX', '321')
  FROM SYSIBM.SYSDUMMY1
```

The result is '123DEFG', '12322XYZ', '' (empty string - all characters removed), and 'XYX123' (no characters removed).

The RTRIM function does not remove instances of '1', '2', and '3' on the left side of the string, before characters that are not '1', '2', or '3'.

Example: **Use the RTRIM function to remove individual characters in the second argument from the end (right side) of the first argument:**

```
SELECT RTRIM ('((-78.0))' , '-0.()')
   FROM SYSIBM.SYSDUMMY1
```

The result is '((-78'.

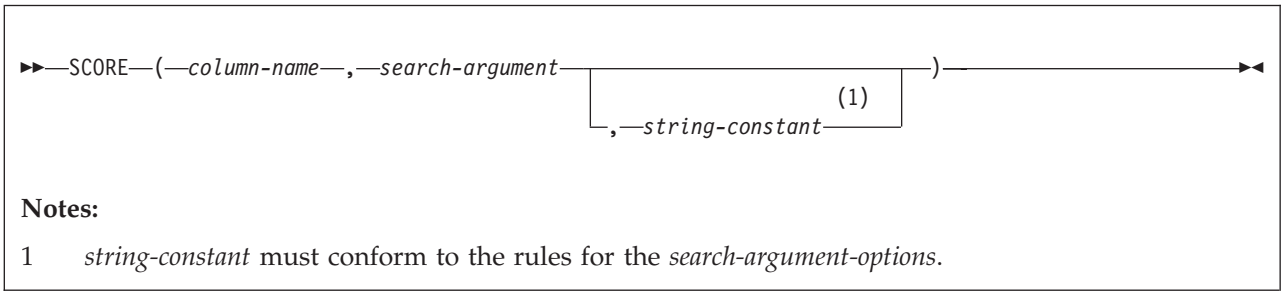
Example: **Use the RTRIM function to remove dollar signs and periods in the second argument from the end (right side) of the first argument:**

```
SELECT RTRIM ('...$VAR$...', '$.')
   FROM SYSIBM.SYSDUMMY1
```

The result is '...$VAR'.

SCORE

The SCORE function searches a text search index using criteria that are specified in a search argument and returns a relevance score that measures how well a document matches the query.



Notes:

1 *string-constant* must conform to the rules for the *search-argument-options*.

search-argument-options:

Notes:

1 The same clause must not be specified more than once.

The schema is SYSIBM.

column-name
Specifies a qualified or unqualified name of a column that has a text search index that is to be searched. The column must exist in the table or view that is identified in the FROM clause in the statement and the column of the table, or the column of the underlying base table of the view must have an associated text search index. The underlying expression of the column of a view must be a simple column reference to the column of an underlying table, either directly or through another nested view.

search-argument
Specifies an expression that returns a value that is a string value (except a LOB) that contains the terms to be searched for and must not be all blanks or the empty string. The actual length of the string must not exceed 4096 Unicode characters. The value is converted to Unicode before it is used to search the text search index. The maximum number of terms per query must not exceed 1024.

string-constant
Identifies a string constant that specifies the search argument options that are in effect for the function.

The options that can be specified as part of the *search-argument-options* are as follows:

QUERYLANGUAGE = *value*
Specifies the query language. The value can be any of the supported language codes. If the QUERYLANGUAGE option is not specified, the default is the language value of the text search index that is used when this function is invoked. If the language value of the text search index is AUTO, the default value for QUERYLANGUAGE is en_US.

RESULTLIMIT = *value*
Specifies the maximum number of results that are to be returned from the underlying search engine. The *value* can be an integer value between 1 and 2 147 483 647. If the RESULTLIMIT option is not specified, no result limit is in effect for the query.

This scalar function cannot be called for each row of the result table, depending on the plan that the optimizer chooses. This function can be called once for the query to the underlying search engine, and a result set of all of the primary keys that match are returned from the search engine. This result set is then joined to the table containing the column to identify the result rows. In this case, the RESULTLIMIT value acts like a FETCH FIRST ?? ROWS from the underlying text search engine and can be used as an optimization. If the search engine is called for each row of the result because the optimizer determines that is the best plan, then the RESULTLIMIT option has no effect.

SYNONYM = *OFF* **or SYNONYM =** *ON*
Specifies whether to use a synonym dictionary that is associated with the text search index. Use the Synonym Tool to add a synonym dictionary to the collection. The default is OFF.

OFF Do not use a synonym dictionary.

ON Use the synonym dictionary that is associated with the text search index.

The result of the function is a double-precision floating-point number. If the second argument can be null, the result can be null. If the second argument is null, the result is the null value. If the third argument is null, the result is as if the third argument was not specified.

The result is greater than 0 but less than 1 if the column contains a match for the search criteria that the search argument specifies. The better a document matches the query, the more relevant the score and the larger the result value. If the column does not contain a match, the result is 0.

SCORE is a non-deterministic function.

Example

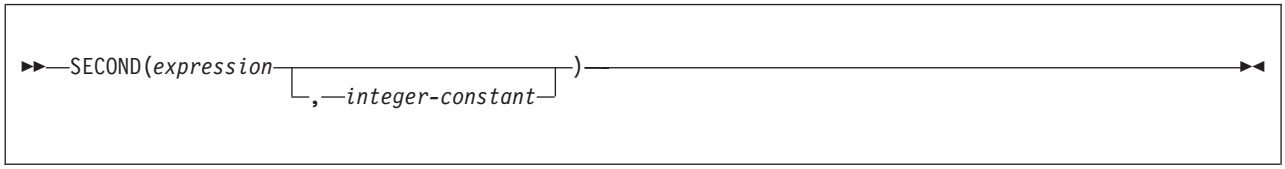
The following statement generates a list of employees in the order of how well their resumes matches the query "programmer AND (java OR cobol)", along with a relevance value that is normalized between 0 (zero) and 100.

```
SELECT EMPNO, INTEGER(SCORE(RESUME, 'programmer AND
 (java OR cobol)') * 100) AS RELEVANCE
 FROM EMP_RESUME
 WHERE RESUME_FORMAT = 'ascii'
 AND CONTAINS(RESUME, 'programmer AND (java OR cobol)') = 1
 ORDER BY RELEVANCE DESC
```

DB2 first evaluates the CONTAINS predicate in the WHERE clause, and therefore, does not evaluate the SCORE function in the SELECT list for every row of the table. In this case, the arguments for SCORE and CONTAINS must be identical.

SECOND

The SECOND function returns the seconds part of a value with optional fractional seconds.



The schema is SYSIBM.

expression
expression must be an expression that returns a value of one of the following built-in data types: a time, a timestamp, a character string, a graphic string, or a numeric data type.

- If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a time or timestamp with an actual length that is not greater than 255 bytes. For the valid formats of string representations of times and timestamps, see "String representations of datetime values" on page 101.
- If *expression* is a number, it must be a time or timestamp duration. For the valid formats of time and timestamp durations, see "Datetime operands" on page 147.

integer-constant
integer-constant must be an integer constant that represents the scale for the fractional seconds portion of *expression*. The value must be in the range 0 through 12. If *integer-constant* is not specified, the result does not include fractional seconds.

If *expression* is a timestamp with a time zone, or a valid string representation of a timestamp with a time zone, the result is determined from the UTC representation of the datetime value.

The result of the function with a single argument is a large integer. The result of the function with two arguments is DECIMAL(2+*s*,*s*) where *s* is the value of *integer-constant*.

The result can be null; if the first argument is null, the result is the null value.

The other rules depend on the data type of the argument:

If the argument is a time, timestamp, or string representation of a time or a timestamp:
The result is the seconds part of the value (0 to 59) and any fractional seconds that are included in the value. If the second argument is specified, the result includes *integer-constant* digits of the fractional seconds part of the value where applicable. If there are no fractional seconds in the value, zeros are returned.

If the argument is a time duration or timestamp duration:
The result is the seconds part of the value (-99 to 99) and any fractional seconds that are included in the value. If the second argument is specified, the result includes *integer-constant* digits of the fractional seconds part of

the value where applicable. If there are no fractional seconds in the value, zeros are returned. A nonzero result has the same sign as the *expression*.

Example 1: Assume that the variable *TIME_DUR* is declared in a PL/I program as DECIMAL(6,0) and can therefore be interpreted as a time duration. When *TIME_DUR* has the value 153045, the following function returns the value 45.

```
SECOND(:TIME_DUR)
```

Example 2: Assume that RECEIVED is a TIMESTAMP column and that one of its values is the internal equivalent of '1988-12-25-17.12.30.000000'. The following function returns the value 30.

```
SECOND(RECEIVED)
```

Example 3: The following invocations of the SECOND function returns the same result:

```
SELECT SECOND('2003-01-02-20.10.05.123456'),
   SECOND('2003-01-02-12.10.05.123456-08:00'),
   SECOND('2003-01-03-05.10.05.123456+09:00')
  FROM SYSIBM.SYSDUMMY1;
```

For each invocation of the SECOND function in this SELECT statement, the result is 5.

When the input argument contains a time zone, the result is determined from the UTC representation of the input value. The string representations of a timestamp with a time zone in the SELECT statement all have the same UTC representation: 2003-01-02-20.10.05.123456. The second portion of the UTC representation is 5.

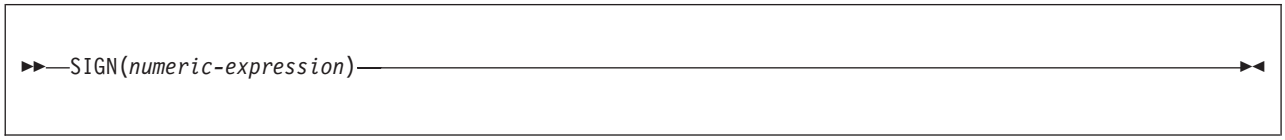
Example 4: Return the seconds with fractional seconds from a current timestamp with milliseconds.

```
SELECT SECOND(CURRENT_TIMESTAMP(3),3)
  FROM SYSIBM.SYSDUMMY1;
```

The SELECT statement returns a DECIMAL(5,3) value that is based on the current timestamp and could be something like `54.321`.

SIGN

The SIGN function returns an indicator of the sign of the argument.



The schema is SYSIBM.

The returned value is one of the following values:
-1 if the argument is less than zero
- 0 if the argument is DECFLOAT negative zero
0 if the argument is zero
1 if the argument is greater than zero

The argument must be an expression that returns a value of any built-in numeric data type, except DECIMAL(31,31).

The argument can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34).

The result has the same data type and length attribute as the argument, except that precision is increased by one if the argument is DECIMAL and the scale of the argument is equal to its precision. For example, an argument with a data type of DECIMAL(5,5) will result in DECIMAL(6,5).

The result can be null; if the argument is null, the result is the null value.

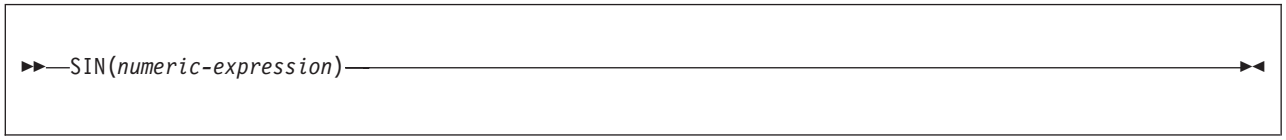
Example: Assume that host variable *PROFIT* is a large integer with a value of 50000.

```
SELECT SIGN(:PROFIT)
  FROM SYSIBM.SYSDUMMY1;
```

This example returns the value 1.

SIN

The SIN function returns the sine of the argument, where the argument is an angle, expressed in radians.



The schema is SYSIBM.

The SIN and ASIN functions are inverse operations.

The argument must be an expression that returns the value of any built-in numeric data type that is not DECFLOAT. If the argument is not a double precision floating-point number, it is converted to one for processing by the function.

The result of the function is a double precision floating-point number.

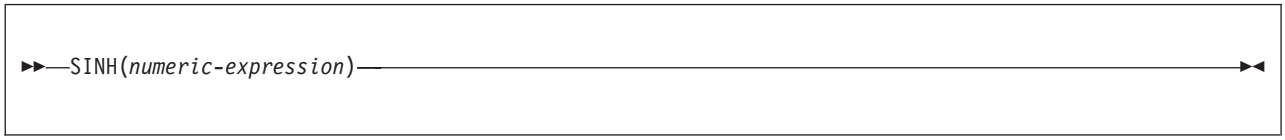
The result can be null; if the argument is null, the result is the null value.

Example: Assume that host variable *SINE* is DECIMAL(2,1) with a value of 1.5. The following statement returns a double precision floating-point number with an approximate value of 0.99.

```
SELECT SIN(:SINE)
  FROM SYSIBM.SYSDUMMY1;
```

SINH

The SINH function returns the hyperbolic sine of the argument, where the argument is an angle, expressed in radians.



The schema is SYSIBM.

The argument must be an expression that returns the value of any built-in numeric data type that is not DECFLOAT. If the argument is not a double precision floating-point number, it is converted to one for processing by the function.

The result of the function is a double precision floating-point number.

The result can be null; if the argument is null, the result is the null value.

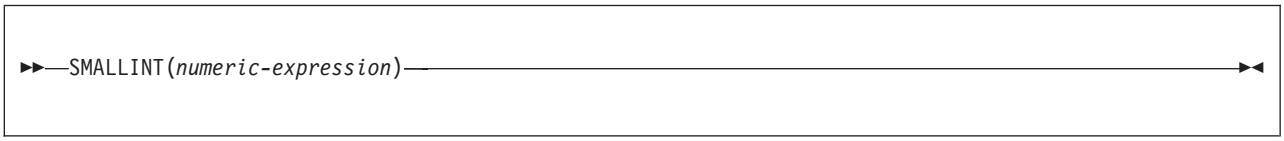
Example: Assume that host variable *HSINE* is DECIMAL(2,1) with a value of 1.5. The following statement returns a double precision floating-point number with an approximate value of 2.12.

```
SELECT SINH(:HSINE)
  FROM SYSIBM.SYSDUMMY1;
```

SMALLINT

The SMALLINT function returns a small integer representation either of a number or of a string representation of a number.

Numeric to Smallint:



String to Smallint:

The schema is SYSIBM.

Numeric to Smallint

numeric-expression
An expression that returns a value of any built-in numeric data type.

The result is the same number that would occur if the argument were assigned to a small integer column or variable. If the whole part of the argument is not within the range of small integers, an error occurs. If present, the decimal part of the argument is truncated.

String to Smallint

string-expression
An expression that returns a value of character or graphic string (except a CLOB or DBCLOB) with a length attribute that is not greater than 255 bytes for a character string or 127 for a graphic string. The string must contain a valid string representation of a number.

The result is the same number that would result from CAST(*string-expression* AS SMALLINT). Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an SQL integer constant. The fractional part of the argument is truncated.

The result of the function is a small integer.

The result can be null; if the argument is null, the result is the null value.

Recommendation: To increase the portability of applications, use the CAST specification. For more information, see "CAST specification" on page 267.

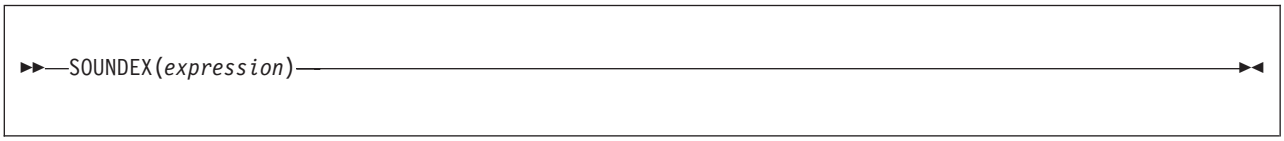
Example: Using sample table DSN8B10.EMP, find the average education level (EDLEVEL) of the employees in department 'A00'. Round the result to the nearest full education level.

```
SELECT SMALLINT(AVG(EDLEVEL)+.5)
  FROM DSN8B10.EMP
  WHERE DEPT = 'A00';
```

Assuming that the five employees in the department have education levels of '19', '18', '14', '18', and '14', the result is '17'.

SOUNDEX

The SOUNDEX function returns a 4-character code that represents the sound of the words in the argument. The result can be compared to the results of the SOUNDEX function of other strings.



The schema is SYSIBM.

expression
An *expression* that must return a value of any built-in numeric, character, or graphic string data type that is not a LOB. A numeric, mixed character, or graphic string value is cast to a Unicode SBCS character string before the function is evaluated. For more information about converting numeric data to a character string, see "VARCHAR" on page 673. For more information about converting mixed or graphic strings to Unicode SBCS, see "CAST specification" on page 267.

The data type of the result is CHAR(4).

The result can be null; if the argument is null, the result is the null value.

The CCSID of the result is the Unicode SBCS CCSID.

The SOUNDEX function is useful for finding strings for which the sound is known but the precise spelling is not. It makes assumptions about the way that letters and combinations of letters sound that can help to search for words with similar sounds. The comparison of words can be done directly or by passing the strings as arguments to the DIFFERENCE function. For more information, see "DIFFERENCE" on page 456.

Example 1: Use the SOUNDEX function to find a row where the sound of the LASTNAME value closely matches the phonetic spelling of 'Loucesy':

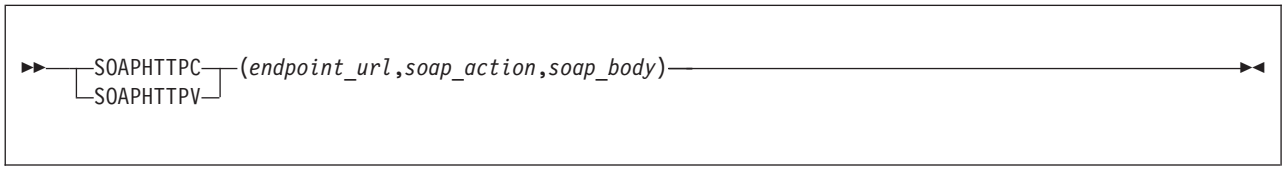
```
SELECT EMPNO, LASTNAME
   FROM DSN910.EMPLOYEE
   WHERE SOUNDEX(LASTNAME) = SOUNDEX('Loucesy');
```

This example returns the following row:

```
000110 LUCCHESSI;
```

SOAPHTTPC and SOAPHTTPV

The SOAPHTTPC function returns a CLOB representation of XML data that results from a SOAP request to the web service that is specified by the first argument. The SOAPHTTPV function returns a VARCHAR representation of XML data that results from a SOAP request to the web service that is specified by the first argument.



The schema is DB2XML.

These functions are deprecated and might not be available in future releases of DB2.

endpoint_url
An expression that returns a value of a built-in character string or graphic string data type that is not a LOB. The value specifies the URL of the web service endpoint for which DB2 is acting as a client.

soap_action
An expression that returns a value of a built-in character string or graphic string data type that is not a LOB. The value specifies a SOAP action URI reference. If it is required for the web service that is specified in *endpoint_url*, the required value is defined in the WSDL of that web service.

soap_body
An expression that returns a value of a built-in character string data type that is defined as VARCHAR(3072) or CLOB(1M). The value specifies the name of an operation with the requested namespace URI, an encoding style, and input arguments. *soap_body* can include well-formed XML content for the SOAP body. The specific operations and arguments for a web service are defined in the WSDL of the specified web service.

If the arguments can be null, the result can be null; if all of the arguments are null, the result is the null value.

The result can be null; if all of the arguments are null, the result is the null value.

Example 1: The following SQL statement retrieves information (as VARCHAR data) about a web service:

```
SELECT DB2XML.SOAPHTTPV(
     'http://www.myserver.com/services/db2sample/ivt.dadx/SOAP',
     'http://tempuri.org/db2sample/ivt.dadx',
     '<testInstallation xmlns="http://tempuri.org/db2sample/ivt.dadx" />')
   FROM SYSIBM.SYSDUMMY1
```

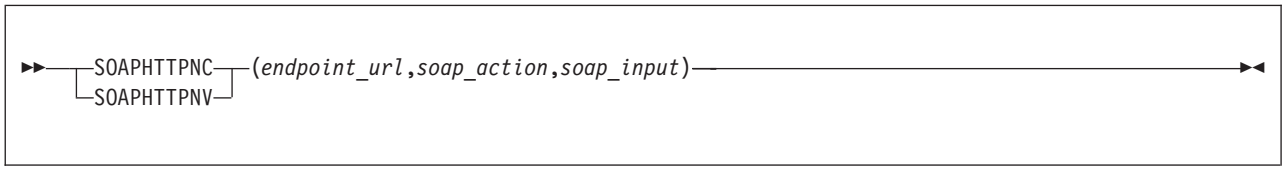
Example 2: The following SQL statement inserts the results (as CLOB data) from a request to a web service into a table:

```
INSERT INTO EMPLOYEE(XMLCOL)
  VALUES (DB2XML.SOAPHTTPC(
     'http://www.myserver.com/services/db2sample/list.dadx/SOAP',
```

```
'http://tempuri.org/db2sample/list.dadx',
'<listDepartments xmlns="http://tempuri.org/db2sample/list.dadx">
    <deptNo>A00</deptNo>
 </listDepartments>'))
```

SOAPHTTPNC and SOAPHTTPNV

The SOAPHTTPNC and SOAPHTTPNV functions allow you to specify a complete SOAP message as input and to return complete SOAP messages from the specified web service. The returned SOAP messages are CLOB or VARCHAR representations of the returned XML data.



The schema is DB2XML.

endpoint_url
Specifies the URL of the web service for which DB2 is acting as a client. *endpoint_url* is defined as a VARCHAR(4096) value. The URL is in the following format:

```
proto://[user[:password]@]hostname[:port]/[path]
```

Where *proto* can be http or https.

soap_action
Specifies a SOAP action URI reference. *soap_action* is defined as a VARCHAR(4096) value. Depending on the web server, *soap_action* might be required. If it is required for the web service that is specified in *endpoint_url*, the required value is defined in the WSDL of that web service.

soap_input
Specifies an XML document that contains the complete SOAP message. *soap_input* can contain optional SOAP headers and must contain a SOAP body that specifies the operation name and parameters to the web service. *soap_input* should be well-formed XML that is defined as VARCHAR(32672) or CLOB(1M).

Example 1: The following SQL statement retrieves information (as VARCHAR data) about a web service:

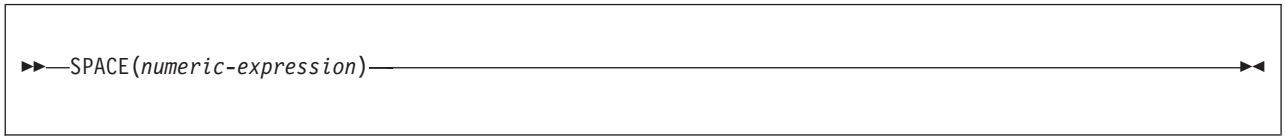
```
SELECT DB2XML.SOAPHTTPNV(
       'http://rpc.geocoder.us/service/soap/',
       '"http://rpc.geocoder.us/Geo/Coder/US#geocode_address"',
       '<?xml version="1.0" encoding="UTF-8" ?>' ||
       '<SOAP-ENV:Envelope ' ||
       'xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" ' ||
       'xmlns:xsd="http://www.w3.org/2001/XMLSchema" ' ||
       'xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">' ||
       '<SOAP-ENV:Body>' ||
       '<ns0:geocode_address ' ||
       'xmlns:ns0="http://rpc.geocoder.us/Geo/Coder/US/" ' ||
       'SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">' ||
       '<address xsi:type="xsd:string">555 Bailey Avenue, San Jose,' ||
       'CA,95141</address>' ||
       '</ns0:geocode_address>' ||
       '</SOAP-ENV:Body>' ||
       '</SOAP-ENV:Envelope>')
  FROM SYSIBM.SYSDUMMY1;
```

Example 2: The following SQL statement inserts the results (as CLOB data) from a request to a web service into a table:

```
INSERT INTO EMPLOYEE(XMLCOL)
  VALUES (DB2XML.SOAPHTTPNC(
     'http://www.myserver.com/services/db2sample/list.dadx/SOAP',
     'http://tempuri.org/db2sample/list.dadx',
     '<?xml version="1.0" encoding="UTF-8" ?>' ||
     '<SOAP-ENV:Envelope ' ||
     'xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" ' ||
     'xmlns:xsd="http://www.w3.org/2001/XMLSchema" ' ||
     'xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">' ||
     '<SOAP-ENV:Body>' ||
     '<listDepartments xmlns="http://tempuri.org/db2sample/list.dadx">
        <deptNo>A00</deptNo>
      </listDepartments>' ||
     '</SOAP-ENV:Body>' ||
     '</SOAP-ENV:Envelope>'))
```

SPACE

The SPACE function returns a character string that consists of the number of SBCS blanks that the argument specifies.



The schema is SYSIBM.

numeric-expression
An expression that returns the value of any built-in integer data type. The expression specifies the number of SBCS blanks for the result, and it must be between 0 and 32764.

The argument can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34) which is then assigned to a BIGINT value.

The result of the function is a varying-length character string (VARCHAR) that contains SBCS data.

If *numeric-expression* is a constant, the length attribute of the result is the constant. Otherwise, the length attribute of the result is 4000. The actual length of the result is the value of *numeric-expression*. The actual length of the result must not be greater than the length attribute of the result.

The result can be null; if the argument is null, the result is the null value.

Example: The following statement returns a character string that consists of 5 blanks followed by a zero-length string.

```
SELECT SPACE(5), SPACE(0)
  FROM SYSIBM.SYSDUMMY1;
```

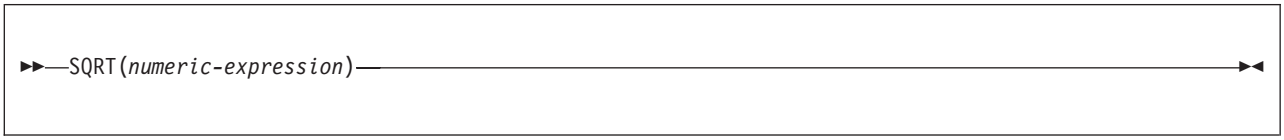
Related concepts:

"Character string encoding schemes" on page 85

"Varying-length character strings" on page 85

SQRT

The SQRT function returns the square root of the argument.



The schema is SYSIBM.

The argument must be an expression that returns the value of any built-in numeric data type. If the argument is DECFLOAT, the operation is performed in DECFLOAT. Otherwise, the argument is converted to a double precision floating-point number for processing by the functions.

The argument can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34).

If the argument is DECFLOAT(*n*), the result is DECFLOAT(*n*). Otherwise, the result of the function is a double precision floating-point number. If the argument is a special decimal floating point value, the general rules for arithmetic operations apply. See "General Arithmetic Operation Rules for DECFLOAT" on page 248 for more information.

The result can be null; if the argument is null, the result is the null value.

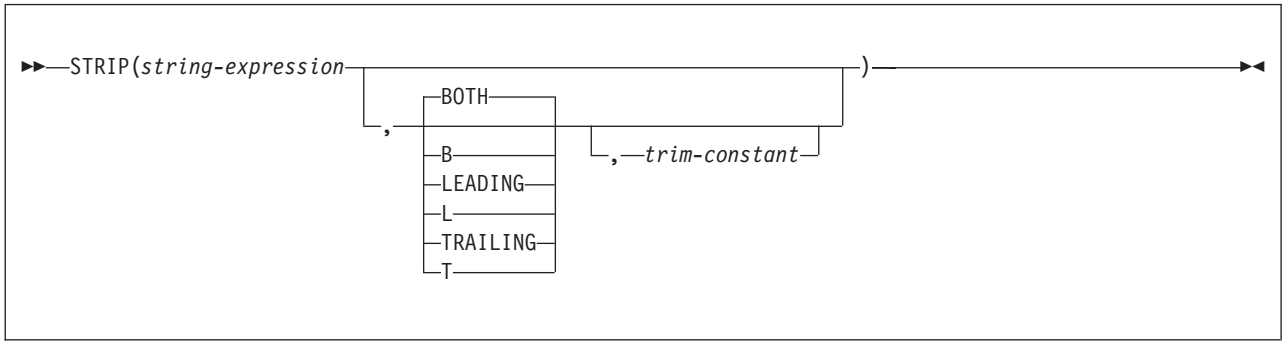
Example: Assume that host variable *SQUARE* is defined as DECIMAL(2,1) and has a value of 9.0. Find the square root of *SQUARE*.

```
SELECT SQRT(:SQUARE)
  FROM SYSIBM.SYSDUMMY1;
```

This example returns a double precision floating-point number with an approximate value of 3.

STRIP

The STRIP function removes blanks or another specified character from the end, the beginning, or both ends of a string expression.



The schema is SYSIBM.

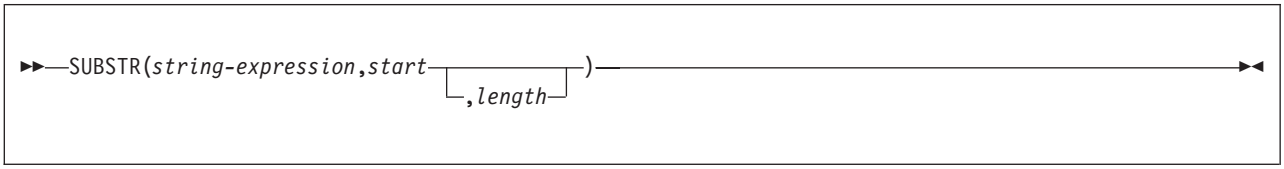
The STRIP function is similar to the TRIM scalar function.

Related reference:

"TRIM" on page 656

SUBSTR

The SUBSTR function returns a substring of a string.



The schema is SYSIBM.

string-expression
An expression that specifies the string from which the result is derived. The string must be a character, graphic, or binary string. If *string-expression* is a character string, the result of the function is a character string. If it is a graphic string, the result of the function is a graphic string. If it is a binary string, the result of the function is a binary string.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.

A substring of *string-expression* is zero or more contiguous characters of *string-expression*. If *string-expression* is a graphic string, a character is a DBCS character. If *string-expression* is a character string or a binary string, a character is a byte. The SUBSTR function accepts mixed data strings. However, because SUBSTR operates on a strict byte-count basis, the result will not necessarily be a properly formed mixed data string.

start
An expression that specifies the position within *string-expression* to be the first character of the result. The value of the large integer must be between 1 and the length attribute of *string-expression*. (The length attribute of a varying-length string is its maximum length.) A value of 1 indicates that the first character of the substring is the first character of *string-expression*.

The argument can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34) which is then assigned to an INTEGER value.

length
An expression that specifies the length of the resulting substring. If specified, *length* must be an expression that returns a value that is a built-in large integer data type.

The argument can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34) which is then assigned to an INTEGER value.The value must be greater than or equal to 0 and less than or equal to *n*, where *n* is the length attribute of *string-expression* - *start* + 1. The specified length must not, however, be the large integer constant 0.

If *length* is explicitly specified, *string-expression* is effectively padded on the right with the necessary number of characters so that the specified substring of *string-expression* always exists. Hexadecimal zeros are used as the padding character when *string-expression* is binary data. Otherwise, a blank is used as the padding character.

If *string-expression* is a fixed-length string, omission of *length* is an implicit specification of `LENGTH(string-expression) - start + 1`, which is the number of characters (or bytes) from the character (or byte) specified by start to the last character (or byte) of *string-expression*. If *string-expression* is a varying-length string, omission of length is an implicit specification of the greater of zero or `LENGTH(string-expression) - start + 1`. If the resulting length is zero, the result is an empty string.

If *length* is explicitly specified by a large integer constant that is 255 or less, and *string-expression* is not a LOB, the result is a fixed-length string with a length attribute of *length*. If *length* is not explicitly specified, but *string-expression* is a fixed-length string and *start* is an integer constant, the result is a fixed-length string with a length attribute equal to `LENGTH(string-expression) - start + 1`. In all other cases, the result is a varying-length string. If *length* is explicitly specified by a large integer constant, the length attribute of the result is *length*; otherwise, the length attribute of the result is the same as the length attribute of *string-expression*.

The result can be null; if any argument is null, the result is the null value.

The CCSID of the result is the CCSID of *string-expression*.

Example 1: FIRSTNME is a VARCHAR(12) column in sample table DSN8B10.EMP. When FIRSTNME has the value 'MAUDE':

```
Function:                Returns:
----------------------------------
SUBSTR(FIRSTNME,2,3)     -- 'AUD'
SUBSTR(FIRSTNME,2)       -- 'AUDE'
SUBSTR(FIRSTNME,2,6)     -- 'AUDE' followed by two blanks
SUBSTR(FIRSTNME,6)       -- a zero-length string
SUBSTR(FIRSTNME,6,4)     -- four blanks
```

Example 2: Sample table DSN8B10.PROJ contains column PROJNAME, which is defined as VARCHAR(24). Select all rows from that table for which the string in PROJNAME begins with 'W L PROGRAM'.

```
SELECT * FROM DSN8B10.PROJ
  WHERE SUBSTR(PROJNAME,1,12) = 'W L PROGRAM ';
```

Assume that the table has only the rows that were supplied by DB2. Then the predicate is true for just one row, for which PROJNAME has the value 'W L PROGRAM DESIGN'. The predicate is not true for the row in which PROJNAME has the value 'W L PROGRAMMING' because, in the predicate's string constant, 'PROGRAM' is followed by a blank.

Example 3: Assume that a LOB locator named *my_loc* represents a LOB value that has a length of 1 gigabyte. Assign the first 50 bytes of the LOB value to host variable *PORTION*.

```
SET :PORTION = SUBSTR(:my_loc,1,50);
```

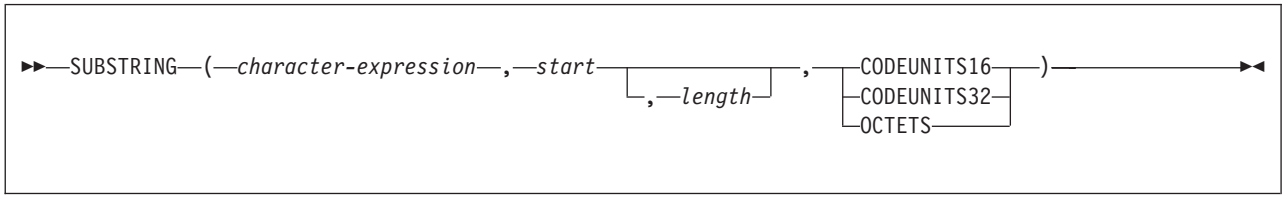
Example 4: Assume that host variable *RESUME* has a CLOB data type and holds an employee's resume. This example shows some of the statements that find the section of department information in the resume and assign it to host variable *DeptBuf*. First, the POSSTR function is used to find the beginning and ending location of the department information. Within the resume, the department information starts with the string 'Department Information Section' and ends immediately before the string 'Education Section'. Then, using these beginning and ending positions, the SUBSTR function assigns the information to the host variable.

```
SET :DInfoBegPos = POSSTR(:RESUME, 'Department Information Section');
SET :DInfoEnPos = POSSTR(:RESUME, 'Education Section');
SET :DeptBuf = SUBSTR(:RESUME, :DInfoBegPos, :DInfoEnPos - :DInfoBegPos);
```

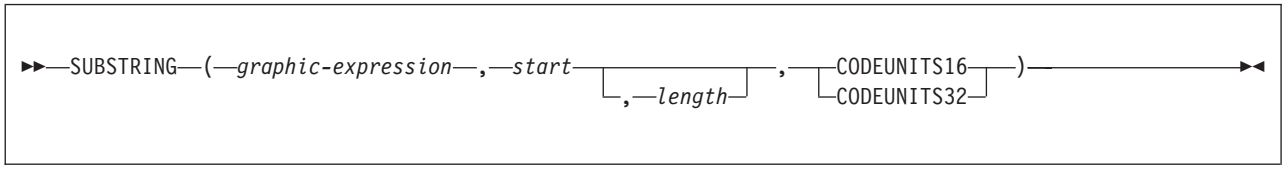
SUBSTRING

The SUBSTRING function returns a substring of a string.

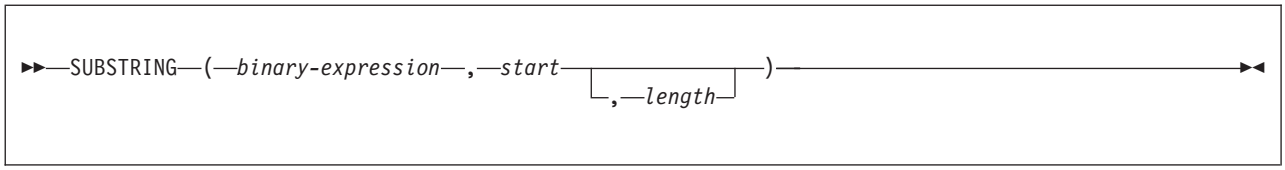
Character:



Graphic:



Binary:



The schema is SYSIBM.

Character

character-expression
An expression that specifies the string from which the result is derived. The string must be a built-in character string.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.The result of the function is a character string.

A substring of *character-expression* is zero or more contiguous units of *character-expression*. If CODEUNITS32 is specified, a unit is a Unicode UTF-32 character. If CODEUNITS16 is specified, a unit is a Unicode UTF-16 character. If OCTETS is specified, a unit is a byte.

start
An expression that specifies the position within the *character-expression* that is to be the first string unit of the result. *start* is expressed in the specified string unit, and must return a large integer value.

The argument can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34) which is then assigned to an INTEGER value.The value of *start* can be positive, negative, or zero. A value of 1 indicates that the first string unit of the result is the first string unit of *character-expression*.

length
An expression that specifies the maximum length of the resulting substring.

If *character-expression* is a fixed-length string, omission of *length* is an implicit specification of CHARACTER_LENGTH(*character-expression*) - *start* + 1, which is the number of string units (CODEUNITS16, CODEUNITS32, or OCTETS) from *start* to the last position of *character-expression*.

If *character-expression* is a varying length string, omission of *length* is an implicit specification of zero or CHARACTER_LENGTH(*character-expression*) - *start* + 1, whichever is greater. If the resulting length is zero, the result is an empty string.

If specified, *length* must be an expression that returns a value that is a built-in large integer data type.

The argument can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34) which is then assigned to an INTEGER value.

The value must be greater than or equal to 0. If a value greater than *n* is specified, where *n* is the length attribute of *character-expression* - *start*+ 1, then *n* is used as the length of the resulting substring. The value is expressed in the units that are explicitly specified.

A rigorous description of the actual length and result: In this description, the term "character" means the "unit specified by string units".

Let *C* be the value of the first argument, let *LC* be the length in characters of *C*, and let *S* be the value of the *start*.

- If *length* is specified, let *L* be the value of *length* and let *E* be *S*+*L*. Otherwise, let *E* be the larger of *LC* + 1 and *S*.
- If either *C*, *S*, or *L* is the null value, the result of the function is the null value.
- If *E* is less than *S*, an exception condition is raised: data exception — substring error.
- Otherwise:
  - If *S* is greater than *LC* or if *E* is less than 1 (one), the result of the function is a zero-length string.
  - Otherwise:
    - Let *S1* be the larger of *S* and 1 (one). Let *E1* be the smaller of *E* and *LC*+1. Let *L1* be *E1*–*S1*.
    - The result of the function is a character string that contains the *L1* characters of *C* starting at character number *S1* in the same order that the characters appear in *C*.

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the string unit that is used to express *start* and *length*. If *character-expression* is a character string that is defined as bit data, CODEUNITS16 and CODEUNITS32 cannot be specified.

CODEUNITS16
Specifies that *start* and *length* are expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32
Specifies that *start* and *length* are expressed in terms of 32-bit UTF-32 code units.

OCTETS
Specifies that *start* and *length* are expressed in terms of bytes.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String unit specifications" on page 87.

Graphic

graphic-expression
An expression that specifies the string from which the result is derived. The string must be a built-in graphic string. The result of the function is a graphic string. A partial surrogate character in the expression is replaced with a blank.

A substring of *graphic-expression* is zero or more contiguous units of *graphic-expression*. If CODEUNITS32 is specified, a unit is a Unicode UTF-32 character. If CODEUNITS16 is specified, a unit is a Unicode UTF-16 character.

start
An expression that specifies the position within the *graphic-expression* that is to be the first string unit of the result. *start* is expressed in the specified string unit, and must return a large integer value. The value of *start* can be positive, negative, or zero. A value of 1 indicates that the first string unit of the result is the first string unit of *graphic-expression*.

The argument can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34) which is then assigned to an INTEGER value.

length
An expression that specifies the maximum length of the resulting substring.

If *graphic-expression* is a fixed-length string, omission of *length* is an implicit specification of `CHARACTER_LENGTH(graphic-expression) - start +1`, which is the number of units (CODEUNITS16, CODEUNITS32) either explicitly or implicitly specified, from the start position to the last position of *graphic-expression*. If *graphic-expression* is a varying length string, omission of *length* is an implicit specification of zero or `CHARACTER_LENGTH(graphic-expression) -start +1`, which is the number of units (CODEUNITS16, CODEUNITS32) either explicitly or implicitly specified, whichever is greater. If the resulting length is zero, the result is an empty string.

If specified, *length* must be an expression that returns a value that is a built-in large integer data type.

The argument can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34) which is then assigned to an INTEGER value.

The value must be greater than or equal to 0. If a value greater than *n* is specified, where *n* is the length attribute of `graphic-expression - start+ 1`, then *n* is used as the length of the resulting substring. The value is expressed in the units that are explicitly specified.

A rigorous description of the actual length and result: In this description, the term "character" means the "unit specified by string units".

Let *C* be the value of the first argument, let *LC* be the length in characters of *C*, and let *S* be the value of the *start*.

- If *length* is specified, let *L* be the value of *length* and let *E* be *S*+*L*. Otherwise, let *E* be the larger of *LC* + 1 and *S*.
- If either *C*, *S*, or *L* is the null value, the result of the function is the null value.
- If *E* is less than *S*, an exception condition is raised: data exception — substring error.
- Otherwise:
  - If *S* is greater than *LC* or if *E* is less than 1 (one), the result of the function is a zero-length string.
  - Otherwise:
    - Let *S1* be the larger of *S* and 1 (one). Let *E1* be the smaller of *E* and *LC*+1. Let *L1* be *E1*–*S1*.
    - The result of the function is a character string that contains the *L1* characters of *C* starting at character number *S1* in the same order that the characters appear in *C*.

CODEUNITS16 or CODEUNITS32
Specifies the string unit that is used to express *start* and *length*.

CODEUNITS16
Specifies that *start* and *length* are expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32
Specifies that *start* and *length* are expressed in terms of 32-bit UTF-32 code units.

For more information about CODEUNITS16 and CODEUNITS32, see "String unit specifications" on page 87.

Binary

binary-expression
An expression that specifies the string from which the result is derived. The string must be a built-in binary string. The result of the function is a binary string.

A substring of *binary-expression* is zero or more contiguous units of *binary-expression*.

start
An expression that specifies the position within *binary-expression* to be the first character of the result. It must be a binary large integer. *start* can be negative or zero. (The length attribute of a varying-length string is its maximum length.) A value of 1 indicates that the first string unit of the substring is the first string unit of *binary-expression*.

length
An expression that specifies the length of the resulting substring.

If *binary-expression* is a fixed-length string, omission of *length* is an implicit specification of CHARACTER_LENGTH(*binary-expression*) - *start* +1, which is the number of units either explicitly or implicitly specified, from the start position to the last position of *binary-expression*. If *binary-expression* is a varying length string, omission of *length* is an implicit specification of zero or CHARACTER_LENGTH(*binary-expression*) -*start* +1, which is the number of

units either explicitly or implicitly specified, whichever is greater. If the resulting length is zero, the result is an empty string.

If specified, *length* must be a value that is a built-in large integer data type. The value must be greater than or equal to 0 and less than or equal to *n*, where *n* is the length attribute of *binary-expression* - *start* + 1. The specified length must not, however, be the large integer constant 0.

A rigorous description of the actual length and result: In this description, the term "character" means the "unit specified by string units".

Let *C* be the value of the first argument, let *LC* be the length in characters of *C*, and let *S* be the value of the *start*.

- If *length* is specified, let *L* be the value of *length* and let *E* be *S*+*L*. Otherwise, let *E* be the larger of *LC* + 1 and *S*.
- If either *C*, *S*, or *L* is the null value, the result of the function is the null value.
- If *E* is less than *S*, an exception condition is raised: data exception — substring error.
- Otherwise:
  - If *S* is greater than *LC* or if *E* is less than 1 (one), the result of the function is a zero-length string.
  - Otherwise:
    - Let *S1* be the larger of *S* and 1 (one). Let *E1* be the smaller of *E* and *LC*+1. Let *L1* be *E1*–*S1*.
    - The result of the function is a character string that contains the *L1* characters of *C* starting at character number *S1* in the same order that the characters appear in *C*.

The data type of the result depends on the data type of the first argument, as shown in the following table.

Table 69. Data type of the result of SUBSTRING

Data type of the first argument	Data type of the result
CHAR or VARCHAR	VARCHAR
CLOB	CLOB If *character-expression* is mixed data, the result is mixed data. Otherwise, the result is SBCS data.
GRAPHIC or VARGRAPHIC	VARGRAPHIC
DBCLOB	DBCLOB
BINARY or VARBINARY	VARBINARY
BLOB	BLOB

The length attribute of the result is equal to the length attribute of the first argument. If CODEUNITS16 or CODEUNITS32 is specified, see "Determining the length attribute of the final result" on page 90 for information about how to calculate the length attribute of the result string.

The result can be null; if any argument is null, the result is the null value.

If the first argument is character or graphic data, the CCSID of the result is the same as that of the first argument.

Example 1: FIRSTNAME is a VARCHAR(12) column in table T1. One of its values is the 6-character string 'Jürgen'. When FIRSTNAME has the value 'Jürgen':

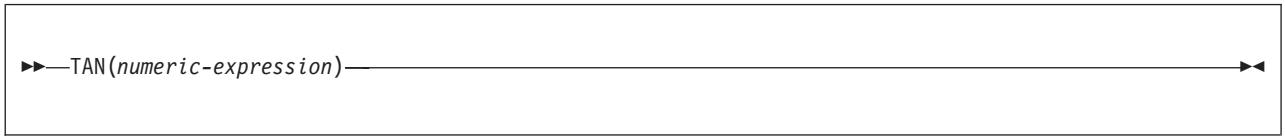
```
Function:                                  Returns:
-----------------------------------------------------------
SUBSTRING(FIRSTNAME,1,2,CODEUNITS32)       'Jü'  -- x'4AC3BC'
SUBSTRING(FIRSTNAME,1,2,CODEUNITS16)       'Jü'  -- x'4AC3BC'
SUBSTRING(FIRSTNAME,1,2,OCTETS)            'J '  -- x'4A20' (a truncated string)
SUBSTRING(FIRSTNAME,8,CODEUNITS16)               -- a zero-length string
SUBSTRING(FIRSTNAME,8,4,OCTETS)                  -- a zero-length string
```

Example 2: C1 is a VARCHAR(12) column in table T1. One of its values is the string 'ABCDEFG'. When C1 has the value 'ABCDEFG':

```
Function:                        Returns:
-------------------------------------------------------
SUBSTRING(C1,-2,2,OCTETS)                -- a zero-length string
SUBSTRING(C1,-2,4,OCTETS)        'A'
SUBSTRING(C1,-2,OCTETS)          'ABCDEFG'
SUBSTRING(C1,0,1,OCTETS)                 -- a zero-length string
```

TAN

The TAN function returns the tangent of the argument, where the argument is an angle, expressed in radians.



The schema is SYSIBM.

The TAN and ATAN functions are inverse operations.

The argument must be an expression that returns the value of any built-in numeric data type that is not DECFLOAT. If the argument is not a double precision floating-point number, it is converted to one for processing by the function.

The result of the function is a double precision floating-point number.

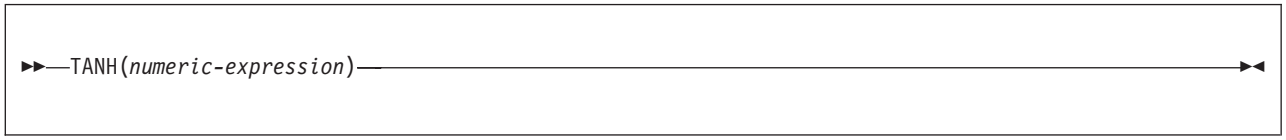
The result can be null; if the argument is null, the result is the null value.

Example: Assume that host variable *TANGENT* is DECIMAL(2,1) with a value of 1.5. The following statement returns a double precision floating-point number with an approximate value of 14.10 .

```
SELECT TAN(:TANGENT)
  FROM SYSIBM.SYSDUMMY1;
```

TANH

The TANH function returns the hyperbolic tangent of the argument, where the argument is an angle, expressed in radians.



The schema is SYSIBM.

The TANH and ATANH functions are inverse operations.

The argument must be an expression that returns the value of any built-in numeric data type that is not DECFLOAT. If the argument is not a double precision floating-point number, it is converted to one for processing by the function.

The result of the function is a double precision floating-point number.

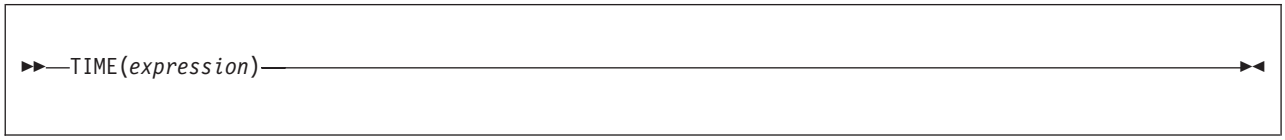
The result can be null; if the argument is null, the result is the null value.

Example: Assume that host variable *HTANGENT* is DECIMAL(2,1) with a value of 1.5. The following statement returns a double precision floating-point number with an approximate value of 0.90.

```
SELECT TANH(:HTANGENT)
  FROM SYSIBM.SYSDUMMY1;
```

TIME

The TIME function returns a time that is derived from a value.



The schema is SYSIBM.

The argument must be an expression that returns a value of one of the following built-in data types: a time, a timestamp, a character string, or a graphic string. If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a time or timestamp with an actual length of not greater than 255 bytes. A time zone in a string representation of a timestamp is ignored. For the valid formats of string representations of times and timestamps, see "String representations of datetime values" on page 101.

If *expression* is a TIMESTAMP WITH TIME ZONE value, *expression* is first cast to TIMESTAMP WITHOUT TIME ZONE, with the same precision as *expression*.

If expression is not a TIME value, *expression* is cast as follows:

- If *expression* is a TIMESTAMP WITH TIME ZONE value, *expression* is cast to TIMESTAMP WITHOUT TIME ZONE, with the same precision as *expression*.
- If *expression* is a string, *expression* is cast to TIME.

The result of the function is a time.

The result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

If the argument is a time
: the result is that time.

If the argument is a timestamp
: the result is the time part of the timestamp.

If the argument is a string
: the result is the time or time part of the timestamp represented by the string. If the CCSID of the string is not the same as the corresponding default CCSID at the server, the string is first converted to that CCSID.

The result CCSID is the appropriate CCSID of the argument encoding scheme and the result subtype is the appropriate subtype of the CCSID.

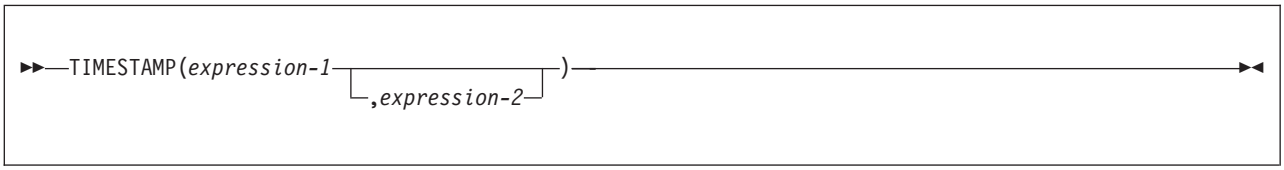
Example: Assume that a table named CLASSES contains one row for each scheduled class. Assume also that the class starting times are in the TIME column named STARTTM. Using these assumptions, select those rows in CLASSES that represent classes that start at 1:30 P.M.

```
SELECT *
  FROM CLASSES
  WHERE TIME(STARTTM) = '13:30:00';
```

TIMESTAMP

The TIMESTAMP function returns a TIMESTAMP WITHOUT TIME ZONE value from its argument or arguments.

See "TIMESTAMP_TZ" on page 645 for a similar function.



The schema is SYSIBM.

The rules for the arguments depend on whether the second argument is specified.

- **If only one argument is specified:**

  The argument must be an expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, a graphic string, or a binary string. If *expression-1* is a character or graphic string, it must not be a CLOB or DBCLOB and it must have one of the following values:

  – A valid string representation of a date or timestamp with an actual length that is not greater than 255 bytes. A time zone in a string representation of a timestamp is ignored. For the valid formats of string representations of timestamps, see "String representations of datetime values" on page 101.

  – A character string or graphic string with an actual length of 8 that is assumed to be a System z® Store Clock value.

  – A character string with an actual length of 13 that is assumed to be a result from the GENERATE_UNIQUE function.

  – A character string or graphic string with an actual length of 14 that represents a valid date and time in the form *yyyyxxddhhmmss*, where *yyyy* is the year, *xx* is the month, *dd* is the day, *hh* is the hour, *mm* is the minute, and *ss* is the seconds.

  If *expression-1* is a binary string, it must not be a BLOB and its value must be one of the following:

  – A binary string with an actual length of 8 bytes that is assumed to be a System z Store Clock value.

  – A binary string with an actual length of 16 bytes that is assumed to be a System z Store Clock extended value.

- **If both arguments are specified:**,

  – If the data type of the second argument is not an integer:

    The first argument must be an expression that returns a value of one of the following built-in data types: a date, a character string, or a graphic string. The second argument must be an expression that returns a value of one of the following built-in data types: a time, a character string, or a graphic string. A character string or graphic string must be a valid string representation of a time.

    If *expression-1* is a character string or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date with an actual length that is not greater than 255 bytes. If *expression-2* is a character string or graphic string, it must not be a CLOB or DBCLOB, and its value

must be a valid string representation of a time with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and times, see "String representations of datetime values" on page 101.

– If the data type of the second argument is integer:

  The first argument must be an expression that returns a value of one of the following built-in data types: a timestamp, a date, a character string, or a graphic string. The second argument must be an integer constant in the range 0 to 12 that represents the timestamp precision.

  If *expression-1* is a character string or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a timestamp or a date with an actual length that is not greater than 255 bytes.

  If *expression-1* is a binary string, it must not be a BLOB, and its value must conform to the rules for when only one argument is specified. The second argument must be an integer constant in the range 0 to 12 that represents the timestamp precision.

The result of the function is a TIMESTAMP WITHOUT TIME ZONE value.

The timestamp precision and other rules depend on whether the second argument is specified:

If both arguments are specified and the second argument is not an integer:
The result is a TIMESTAMP(6) WITHOUT TIME ZONE value with the date that is specified by the first argument and the time that is specified by the second argument. The fractional seconds part of the timestamp is zero.

If both arguments are specified and the second argument is an integer:
The result is a TIMESTAMP WITHOUT TIME ZONE value with the precision that is specified in the second argument.

If only one argument is specified and it is a TIMESTAMP (*p*) WITHOUT TIME ZONE:
The result is that TIMESTAMP (*p*) WITHOUT TIME ZONE value.

If only one argument is specified and it is a TIMESTAMP(*p*) WITH TIME ZONE:
The result is the argument value, cast to TIMESTAMP(*p*) WITHOUT TIME ZONE. The value is the local timestamp, not UTC.

If only one argument is specified and it is a date:
The result is that date with an assumed time of midnight that is cast to TIMESTAMP(0) WITHOUT TIME ZONE.

If only one argument is specified and it is a character or graphic string:
The result is the TIMESTAMP(6) WITHOUT TIME ZONE value that is represented by that string extended with any missing time information. If the argument is a string of length 14, the TIMESTAMP has a fractional seconds part of zero. The string value must not contain a specification of time zone.

If only one argument is specified and it is a binary string:
The result is the TIMESTAMP(6) WITHOUT TIME ZONE value that is represented by that string. If the year value in the resulting timestamp is greater than 9999 an error is returned (SQLSTATE 22007, SQLCODE -180).

If the arguments include only date information, the time information in the result value is all zeros.

The result can be null; if any argument is null, the result is the null value.

If an argument is a string with a CCSID that is not the same as the corresponding default CCSID at the server, the string is first converted to that CCSID.

The result CCSID is the appropriate CCSID of the argument encoding scheme and the result subtype is the appropriate subtype of the CCSID. If both arguments are specified and their encoding schemes are different, the result CCSID is the appropriate CCSID of the application encoding scheme.

Syntax alternatives: If only one argument is specified, the CAST specification should be used for maximal portability. For more information, see "CAST specification" on page 267.

Example 1: Assume that table TABLEX contains a DATE column named DATECOL and a TIME column named TIMECOL. For some row in the table, assume that DATECOL represents 25 December 2008 and TIMECOL represents 17 hours, 12 minutes, and 30 seconds after midnight. The following function returns the value '2008-12-25-17.12.30.000000'.

```
TIMESTAMP(DATECOL, TIMECOL)
```

Example 2: Assume that host variable *PRSTSZ* contains '2008-02-29.20.00.000000 -08.30. The following statement returns the value '2008-02-29.20.00.000000':

```
SELECT TIMESTAMP(:PRSTSZ)
 FROM PROJECT;
```

Example 3: The following invocation of the TIMESTAMP function converts a timestamp string with 7 digits of fractional seconds to a TIMESTAMP(9) WITHOUT TIME ZONE value and returns a value of '2007-09-24-15.53.37.216247400':

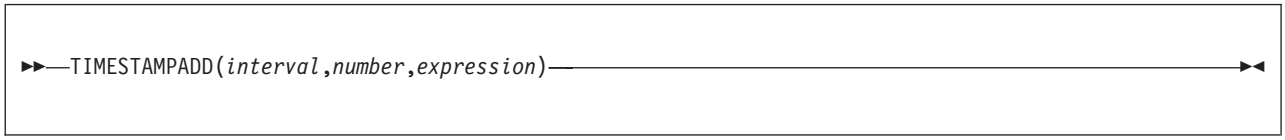
```
TIMESTAMP('2007-09-24-15.53.37.2162474',9);
```

Specifying an LRSN as an argument

When a 6-byte LRSN is used as the argument to the TIMESTAMP function, it must be left justified and padded on the right to a total length of 8 bytes. When a 10-byte LRSN is used, it must be left justified and padded on the right to a total length of 16 bytes.

TIMESTAMPADD

The TIMESTAMPADD function returns the result of adding the specified number of the designated interval to the timestamp value.



The schema is SYSIBM.

interval
An expression that returns a value of a built-in SMALLINT or INTEGER data type. The following values are valid values for *interval*:

Table 70. Valid values for intervals

Valid values for *interval*	**equivalent intervals**
1	Microseconds
2	Seconds
4	Minutes
8	Hours
16	Days
32	Weeks
64	Months
128	Quarters
256	Years

number
An expression that returns a value of a built-in SMALLINT or INTEGER data type.

expression
An expression that returns a value of a built-in TIMESTAMP WITHOUT TIME ZONE data type.

The result of the function is the same timestamp data type with the same timestamp precision as *expression*.

The result can be null; if any argument is null, the result is the null value.

The result is determined using the normal rules for datetime arithmetic. See "Datetime arithmetic in SQL" on page 257. When the interval to add is expressed as weeks, the result is calculated as if *number* `x 7 days` had been specified. When the interval to add is expressed as quarters, the result is calculated as if *number* `x 3 months` had been specified.

Example 1: The following example will add 40 years to the specified timestamp. An interval of 256 designates years, while 40 specifies the number of intervals to add. The following statement returns the value '2005-07-27-15.30.00.000000'.

```
SELECT TIMESTAMPADD(256,40,TIMESTAMP('1965-07-27-15.30.00'))
  FROM SYSIBM.SYSDUMMY1;
```

Example 2: The following example will add 18 months to the specified timestamp. An interval of 64 designates months, while 18 specifies the number of intervals to add. The following statement returns the value '2008-07-20-08.08.00.000000'.

```
SELECT TIMESTAMPADD(64,18,TIMESTAMP('2007-01-20-08.08.00'))
  FROM SYSIBM.SYSDUMMY1;
```

Example 3: The following example will subtract 16 quarters (4 years) from the specified timestamp. An interval of 128 designates quarters, while -16 specifies the number of intervals to add (the '-' adds a negative amount). The following statement returns the value '2003-09-28-05.30.00.000000'.

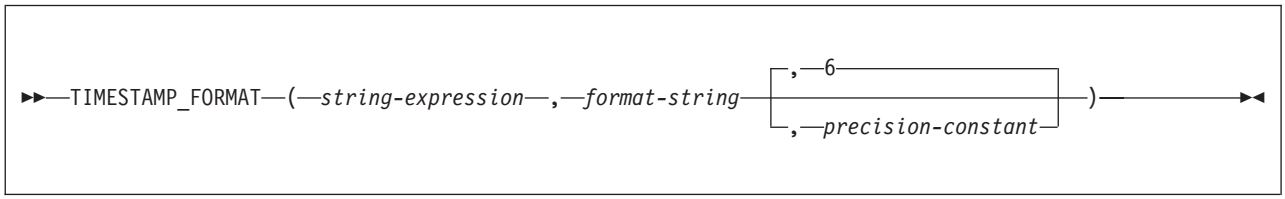
```
SELECT TIMESTAMPADD(128,-16,TIMESTAMP('2007-09-28-05.30.00'))
   FROM SYSIBM.SYSDUMMY1;
```

Example 4: The following example will add 18 weeks to the specified timestamp. An interval of 32 designates weeks, while 18 specifies the number of intervals to add. The following statement returns the value '2007-05-27-08.08.00.000000'.

```
SELECT TIMESTAMPADD(32,18,TIMESTAMP('2007-01-20-08.08.00'))
 FROM SYSIBM.SYSDUMMY1;
```

TIMESTAMP_FORMAT

The TIMESTAMP_FORMAT function returns a TIMESTAMP WITHOUT TIME ZONE value that is based on the interpretation of the input string using the specified format.



The schema is SYSIBM.

string-expression
An expression that returns a value of any built-in character or graphic string data type, other than a CLOB or DBCLOB, with a length attribute that is not greater than 255 bytes. The *string-expression* must contain the components of a timestamp that correspond to the format that is specified in *format-string*, except for hour, minute, second, or fractional seconds.

format-string
The expression must return a value that is a built-in character or graphic string data type, other than a CLOB or DBCLOB, with a length attribute that is not greater than 255 bytes. The actual length must not be greater than 255 bytes. The value is a template for how *string-expression* is interpreted and then converted to a timestamp value.

A valid *format-string* must contain at least one format element, must not contain multiple specifications for any component of a timestamp, and can contain any combination of the format elements, unless otherwise noted in the following table. For example, *format-string* cannot contain both YY and YYYY, because both are used to interpret the year component of a *string-expression*. Two format elements can be separated by one or more of the following separator characters:

- minus sign (-)
- period (.)
- forward slash (/)
- comma (,)
- apostrophe (')
- semicolon (;)
- colon (:)
- blank ( )

Separator characters can also be specified at the start or end of *format-sting*. These separator characters can be used in any combination in the format string, for example 'YYYY/MM-DD HH:MM.SS'. Separator character that is specified in a *string-expression* are used to separate components and are not required to match the separator character that is specified in the *format-string*.

Table 71. Format elements for the TIMESTAMP_FORMAT function

Format element	Related component of a timestamp	Description
AM or PM [1]	hour	Meridian indicator (morning or evening) without periods. This format element uses the exact strings "AM" or "PM".
A.M. or P.M. [1]	hour	Meridian indicator (morning or evening) with periods. This format element uses the exact strings "A.M." or "P.M."
D [1]	none	Day of the week (1-7).
DD	day	Day of the month (0-31).
DDD	month, day	Day of the year (001-366).
FF or FF*n*	fractional seconds	Fractional seconds (0-999999999999). The number *n* is used to specify the number of digits that is expected in the *string-expression*. Valid values for *n* are 1-12 with no leading zeros. Specifying FF is equivalent to specifying FF6. When the component in *string-expression* that corresponds to the FF format element is followed by a separator character or is the last component, the number of digits for the fractional seconds can be less than what is specified by the format element. In this case, zero digits are padded onto the right of the number of specified digits.
HH	hour	HH behaves the same as HH12.
HH12	hour	Hour of the day (01-12) in 12-hour format. AM is the default meridian indicator.
HH24	hour	Hour of the day (00-24) in 24-hour format.
J	year, month, and day	Julian day (number of days since January 1, 4713 BC).
MI	minute	Minute (00-59).
MM	month	Month (01-12).
MONTH, Month, or month [1, 2]	month	Name of the month in English.
MON, Mon, or mon [1, 2]	month	Abbreviated name of the month in English.

Table 71. Format elements for the TIMESTAMP_FORMAT function (continued)

Format element	Related component of a timestamp	Description
NNNNNN	microseconds	Microseconds (000000-999999).
RR	year	Last two digits of the adjusted year (00-99).
RRRR	year	Four digit adjusted year (0000-9999).
SS	seconds	Seconds (00-59).
SSSSS	hours, minutes, and seconds	Seconds since the previous midnight (00000 - 86400).
Y	year	Last digit of the year (0-9). First three digits of the current year are used to determine the full 4-digit year.
YY	year	Last two digits of the year (00-99). First two digits of the current year are used to determine the full 4-digit year.
YYY	year	Last three digits of the year (000-999). First digit of the current year is used to determine the full 4-digit year.
YYYY	year	4-digit year (0000-9999).

Notes:

1. This format element is case sensitive.
2. Only these exact spellings and case combinations can be used. If this format element is specified in an invalid case combination an error is returned.
3. The D format element does not contribute to any components of the resulting timestamp. However, a specified value for this format element must be correct for the combination of the day component of the resulting timestamp. For example, a value of '5' for *string-expression* is valid for a format string value of 'D'. However, value of '9' for *string-expression* would result in an error for the same *format-string*.

The RR and RRRR format elements can be used to change how a specification for a year is to be interpreted by adjusting the value to produce a 2-digit or a 4-digit value depending on the leftmost two digits of the current year according to the following table:

Table 72. Correspondence of adjusted year value and timestamp component

Digits of the current year	Two-digit year in *string-expression*	First two digits of the year component of timestamp
00-50	00-49	First two digits of the current year
51-99	00-49	First two digits of the current year + 1
00-50	50-99	First two digits of the current year -1

Table 72. Correspondence of adjusted year value and timestamp component (continued)

Digits of the current year	Two-digit year in *string-expression*	First two digits of the year component of timestamp
51-99	50-99	First two digits of the current year

For example, if the current year is 2007, '86' with format 'RR' means 1986, but if the current year is 2052, it means 2086.

The following defaults are used when a format-string does not include a format element for one of the following components of a timestamp:

Timestamp component	Default
year	current year, as 4 digits
month	current month. as 2 digits
day	01 (first day of the month)
hour	00
minute	00
second	00
fractional seconds	a number of zeros to match the timestamp precision of the result

If *string-expression* does not include a value that corresponds to an hour, minute, second, or fractional seconds format element that is specified in the *format-string*, the same defaults are used.

Leading zeros can be specified for any component of the timestamp value (that is, month, day, hour, minutes, seconds) that does not have the maximum number of significant digits for the corresponding format element in the *format-string*.

A substring of the *string-expression* that represents a component of a timestamp (such as year, month, day, hour, minutes, seconds) can include fewer than the maximum number of digits for that component of the timestamp that is indicated by the corresponding format element. Any missing digits default to zero. For example, with a format-string of 'YYYY-MM-DD HH24:MI:SS', an input value of '999-3-9 5:7:2' produces the same result as '0999-03-09 05:07:02'.

precision-constant
An integer constant that specifies the timestamp precision of the result. The value must be in the range 0 to 12. If *precision-constant* is not specified, the timestamp precision defaults to 6.

The result of the function is a TIMESTAMP with a precision that is based on *precision-constant*.

If either of the first two arguments can be null, the result can be null; if either of the first two arguments is null, the result is the null value.

The result CCSID is the appropriate CCSID of the encoding scheme of the first argument and the result subtype is the appropriate subtype of the CCSID.

Notes

Julian and Gregorian calendar:
The transition from the Julian calendar to the Gregorian calendar on 15 October 1582 is taken into account by this function.

Determinism:
TIMESTAMP_FORMAT is a deterministic function. However, the following invocations of the function depend on the value of the special register CURRENT TIMESTAMP.

- *format-string* is not a constant
- *format-string* is a constant and includes format elements that are locale sensitive
- *format-string* is a constant and does not include a format element that fully defines the year (that is, J or YYYY). In this case the current year is used.
- *format-string* is a constant and does not include a format element that fully defines the month (for example, J, MM, MONTH, or MON). In this case the current month is used.

These invocations, which depend on the value of a special register, cannot be used wherever special registers cannot be used.

Using the 'D', 'Y', and 'y' format elements:
DB2 for z/OS does not support the 'DY', 'dy', and 'Dy' format elements that are supported by other platforms. If 'DY' or 'Dy' is specified in the format string, it is interpreted as the 'D' format element followed by the 'Y' or 'y' format element. This behavior might change in a future release. To ensure that a 'D' followed by 'Y' or 'y' is interpreted as two separate format elements, include a separator character after the 'D' format element.

Syntax alternatives:
TO_DATE can be specified as a synonym for **TIMESTAMP_FORMAT**.

Example 1:
Insert a row into the IN_TRAY table with a receiving timestamp that is equal to one second before the beginning of the year 2000 (December 31, 1999 at 23:59:59).

```
INSERT INTO IN_TRAY (RECEIVED)
 VALUES (TIMESTAMP_FORMAT('1999-12-31 23:59:59', 'YYYY-MM-DD HH24:MI:SS'))
```

Example 2:
An application receives strings of date information into a variable called *INDATEVAR*. This value is not strictly formatted and might include two or four digits for years, and one or two digits for months and days. Date components might be separated with minus sign (-) or forward-slash (/) characters and are expected to be in day, month, and year order. Time information consists of hours (in 24-hour format) and minutes, and is usually separated by a colon. Sample values include '15/12/98 13:48' and '9-3-2004 8:02'. Insert such values into the IN_TRAY table.

```
INSERT INTO IN_TRAY (RECEIVED)
 VALUES (TIMESTAMP_FORMAT(:INDATEVAR, 'DD/MM/RRRR HH24:MI'))
```

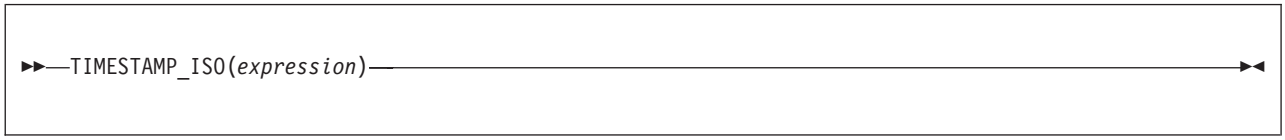
The use of 'RRRR' in the format allows for 2-digit and 4-digit year values and assigns the missing first two digits based on the current year. If 'YYYY' is used, input values with a 2-digit year will have leading zeros. The

forward-slash separator also allows the minus sign character. Assuming a current year of 2007, resulting timestamp values from the sample values are as follows:

```
'15/12/98 13:48' --> 1998-12-15-13.48.00.000000
'9-3-2004 8:02' --> 2004-03-09-08.02.00.000000
```

TIMESTAMP_ISO

The TIMESTAMP_ISO function returns a timestamp value that is based on a date, a time, or a timestamp argument.



The schema is SYSIBM.

If the argument is a date, TIMESTAMP_ISO inserts a value of zero for the time and the partial seconds parts of the timestamp. If the argument is a time, TIMESTAMP_ISO inserts the value of CURRENT DATE for the date part of the timestamp and a value of zero for the partial seconds part of the timestamp.

expression
An expression that returns a value of one of the following built-in data types:

- a TIMESTAMP WITHOUT TIME ZONE
- a date
- a time
- a character string
- or a graphic string

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB and its value must be a valid string representation of a date, a time, or a timestamp. For the valid formats of string representations of dates, times, and timestamps, see "String representations of datetime values" on page 101.

If *expression* is a timestamp, the result of the function is the same timestamp data type with the same precision as *expression*. Otherwise, the result of the function is a TIMESTAMP (6) WITHOUT TIME ZONE.

The result can be null; if the argument is null, the result is the null value.

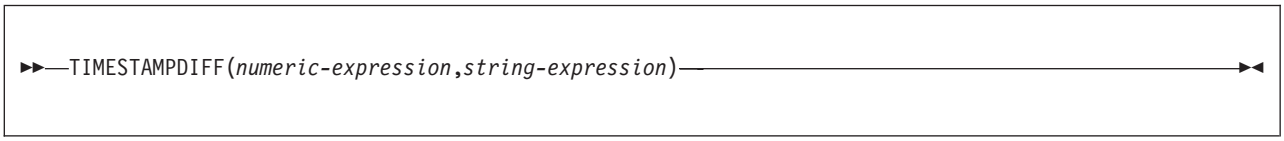
Recommendation: Use the CAST specification for maximum portability. For more information, see "CAST specification" on page 267.

Example: Assume the following date value '1965-07-27'. The following statement returns the value '1965-07-27-00.00.00.000000'.

```
SELECT TIMESTAMP_ISO( DATE( '1965-07-27' ) )
  FROM SYSIBM.SYSDUMMY1
```

TIMESTAMPDIFF

The TIMESTAMPDIFF function returns an estimated number of intervals of the type that is defined by the first argument, based on the difference between two timestamps.



The schema is SYSIBM.

numeric-expression

An expression that returns a value that is a built-in SMALLINT or INTEGER data type. The value specifies the interval that is used to determine the difference between two timestamps. The following table lists the valid values for *numeric-expression*:

Table 73. Valid values for numeric-expression and equivalent intervals that are used to determine the difference between two timestamps

Valid values for *numeric-expression*	**equivalent intervals**
1	Microseconds
2	Seconds
4	Minutes
8	Hours
16	Days
32	Weeks
64	Months
128	Quarters
256	Years

string-expression

An expression that returns a value of a built-in character string or a graphic string data type that is not a LOB. The value is expected to be the result of subtracting two timestamps and converting the result to a character string of length 22. The string value must not have more than 6 digits to the right of a decimal point. If the supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The following table describes the elements of *string-expression*:

Table 74. TIMESTAMPDIFF String Elements

String elements	**Valid values**	**Character position from the decimal point (negative is left)**
Years	1-9998 or blank	-14 to -11
Months	0-11 or blank	-10 to -9
Days	0-30 or blank	-8 to -7
Hours	0-24 or blank	-6 to -5

Table 74. TIMESTAMPDIFF String Elements (continued)

String elements	Valid values	Character position from the decimal point (negative is left)
Minutes	0-59 or blank	-4 to -3
Seconds	0-59	-2 to -1
Decimal separator	period	0
Microsecond	000000-999999	1 to 6

The result of the function is an integer with the same sign as the second argument.

The result can be null; if any argument is null, the result is the null value.

The returned value is determined for each interval as indicated by the following table:

Table 75. TIMESTAMPDIFF Computations

Result interval	Computation using duration elements
Years	years
Quarters	integer value of (months+(years*12))/3
Months	months + (years*12)
Weeks	integer value of ((days+(months*30))/7)+(years*52)
Days	days + (months*30)+(years*365)
Hours	hours+ ((days + (months*30)+(years*365))*24)
Minutes (the absolute value of the duration must not exceed 40850913020759.999999)	minutes + (hours+((days+(months*30)+(years*365))*24))*60
Seconds (the absolute value of the duration must be less than 680105031408.000000)	seconds + (minutes+(hours+((days+(months*30) +(years*365))*24))*60)*60
Microseconds (the absolute value of the duration must be less than 3547.483648)	microseconds + (seconds+(minutes*60))*1000000

The following assumptions are used in estimating a difference:

- One year has 365 days
- One year has 52 weeks
- One year has 12 months
- One month has 30 days
- One day has 24 hours
- One hour has 60 minutes
- One minute has 60 seconds

The use of these assumptions imply that some result values are an estimate of the interval. Consider the following examples:

- Difference of 1 month where the month has less than 30 days.

  ```
  TIMESTAMPDIFF(16, CHAR(TIMESTAMP('1997-03-01-00.00.00')
    - TIMESTAMP('1997-02-01-00.00.00')))
  ```

  The result of the timestamp arithmetic is a duration of 00000100000000.000000, or 1 month. When the TIMESTAMPDIFF function is invoked with 16 for the interval argument (days), the assumption of 30 days in a month is applied and the result is 30.

- Difference of 1 day less than 1 month where the month has less than 30 days.

  ```
  TIMESTAMPDIFF(16, CHAR(TIMESTAMP('1997-03-01-00.00.00')
    - TIMESTAMP('1997-02-02-00.00.00')))
  ```

  The result of the timestamp arithmetic is a duration of 00000027000000.000000, or 27 days. When the TIMESTAMPDIFF function is invoked with 16 for the interval argument (days), the result is 27.
- Difference of 1 day less than 1 month where the month has 31 days.

  ```
  TIMESTAMPDIFF(64, CHAR(TIMESTAMP('1997-09-01-00.00.00')
    - TIMESTAMP('1997-08-02-00.00.00')))
  ```

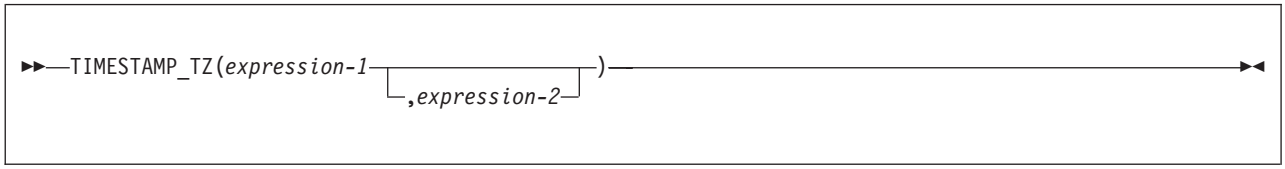
  The result of the timestamp arithmetic is a duration of 00000030000000.000000, or 30 days. When the TIMESTAMPDIFF function is invoked with 64 for the interval argument (months), the result is 0. The days portion of the duration is 30, but it is ignored because the interval specified months.

Example: The following statement estimates the age of employees in months and returns that value as AGE_IN_MONTHS:

```
SELECT
  TIMESTAMPDIFF(64, CAST(CURRENT_TIMESTAMP-CAST(BIRTHDATE AS TIMESTAMP)
                AS CHAR(22)))
    AS AGE_IN_MONTHS
  FROM EMPLOYEE;
```

TIMESTAMP_TZ

The TIMESTAMP_TZ function returns a TIMESTAMP WITH TIME ZONE value from the input arguments.



The schema is SYSIBM.

expression-1
An expression that returns a value of one of the following built-in data types:

- a timestamp without time zone
- a timestamp with time zone
- a character string
- a graphic string

If *expression-1* is a character string or a graphic string, it must conform to the following rules:

- It must not be a CLOB or DBCLOB
- Its value must be a valid string representation of a timestamp without a time zone or a timestamp with a time zone value
- It must have an actual length that is not greater than 255 bytes

For the valid formats of string representations of datetime values, see "String representations of datetime values" on page 101.

If *expression-2* is specified, *expression-1* must be a timestamp without a time zone, or a string representation of a timestamp without a time zone.

expression-2
An expression that returns a character string or a graphic string.

If *expression-2* is a character string or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a time zone in the format of '±*th*:*tm*' with values ranging from -12:59 to +14:00, where *th* represents time zone hour and *tm* represents time zone minute.

The result of the function is equivalent to invoking the CAST specification, as indicated in the following table:

Table 76. TIMESTAMP_TZ function and equivalent CAST specification

TIMESTAMP_TZ function syntax	Equivalent CAST specification syntax
TIMESTAMP_TZ(*timestamp_wo_tz*)	CAST(*timestamp_wo_tz* AS TIMESTAMP WITH TIME ZONE)
TIMESTAMP_TZ(*timestamp_wo_tz*, *n*)	CAST(*timestamp_wo_tz* AS TIMESTAMP(*n*) WITH TIME ZONE)
TIMESTAMP_TZ(*timestamp_wo_tz*, *timezone*)	CAST(CONCAT(VARCHAR(*timestamp_wo_tz*, *timezone*) AS TIMESTAMP WITH TIME ZONE)
TIMESTAMP_TZ(*timestamp_wo_tz*, *timezone*, *n*)	CAST(CONCAT(VARCHAR(*timestamp_wo_tz*, *timezone*) AS TIMESTAMP(*n*) WITH TIME ZONE)

Table 76. TIMESTAMP_TZ function and equivalent CAST specification (continued)

TIMESTAMP_TZ function syntax	Equivalent CAST specification syntax
TIMESTAMP_TZ(*timestamp_w_tz*)	CAST(*timestamp_w_tz* AS TIMESTAMP WITH TIME ZONE)
TIMESTAMP_TZ(*timestamp_w_tz*, *n*)	CAST(*timestamp_w_tz* AS TIMESTAMP(*n*) WITH TIME ZONE)
TIMESTAMP_TZ(*timestamp_w_tz*, *timezone*)	N/A
TIMESTAMP_TZ(*timestamp_w_tz*, *timezone*, *n*)	N/A

timestamp_wo_tz
A timestamp without time zone value.

timestamp_w_tz
A timestamp with time zone value.

timezone
A time zone value.

n The precision value.

When a string representation of a timestamp is a single-byte character set (SBCS) with a CCSID that is not the same as the default CCSID for SBCS data, that value is converted to the default CCSID for SBCS data before it is interpreted and converted to a timestamp value.

Syntax alternatives:

- If only one argument is specified, the CAST specification should be used to ensure maximal portability. For more information, see "CAST specification" on page 267
- FROM_TZ can be specified as a synonym for TIMESTAMP_TZ when TIMESTAMP_TZ specifies both *expression-1* and *expression-2*.

Example 1: Assume that *TIMES* is a host variable with the value 2008-02-29-20.00.00.000000 and that *TZ* is a host variable with the value -3.00. Convert the value of *TIMES* and *TZ* to a timestamp with time zone.

```
SET :TIMESZ = TIMESTAMP_TZ(:TIMES, :TZ);
```

The host variable *TIMESZ* is set with the value that represents the timestamp with time zone as 2008-02-29-20.00.00.000000 -03.00.

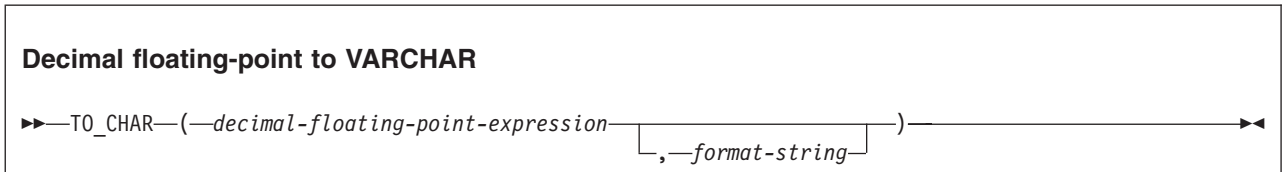
TO_CHAR

The TO_CHAR function returns a character string representation of a timestamp value that has been formatted using a specified character template.

Character to VARCHAR

Timestamp to VARCHAR

Decimal floating-point to VARCHAR

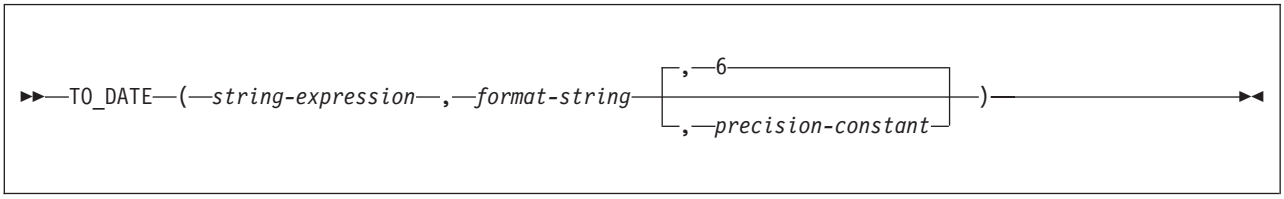


The schema is SYSIBM.

The TO_CHAR scalar function is a synonym for the VARCHAR_FORMAT scalar function.

TO_DATE

The TO_DATE function returns a timestamp value that is based on the interpretation of the input string using the specified format.

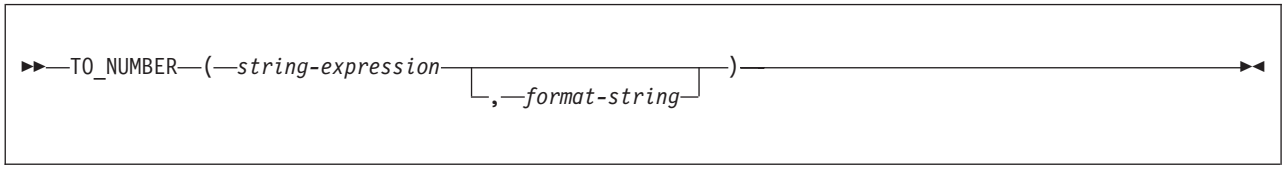


The schema is SYSIBM.

The TO_DATE scalar function is a synonym for the TIMESTAMP_FORMAT scalar function.

TO_NUMBER

The TO_NUMBER function returns a DECFLOAT(34) value that is based on the interpretation of the input string using the specified format.

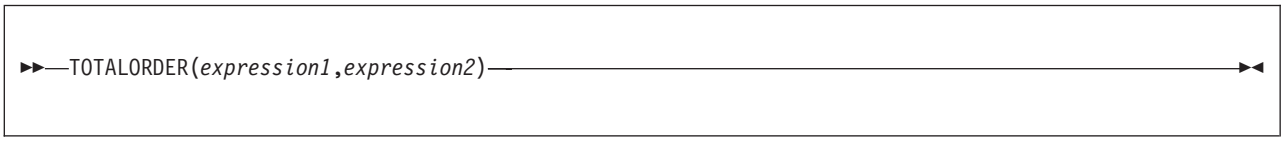


The schema is SYSIBM.

The TO_NUMBER scalar function is a synonym for the DECFLOAT_FORMAT scalar function.

TOTALORDER

The TOTALORDER function returns an ordering for DECFLOAT values. The TOTALORDER function returns a small integer value that indicates how *expression1* compares with *expression2*.



The schema is SYSIBM.

expression1
An expression that returns a built-in DECFLOAT value.

The argument can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34).

expression2
An expression that returns a built-in DECFLOAT value.

The argument can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34).

Numeric comparison is exact, and the result is determined for finite operands as if range and precision are unlimited. An overflow or underflow conditions cannot occur.

If one value is DECFLOAT(16) and the other is DECFLOAT(34), the DECFLOAT(16) value is converted to DECFLOAT(34) before the comparison is made.

TOTALORDER determines ordering based on the total order predicate rules of IEEE 754R, with the following result:

- -1 if the first argument is lower in order compared to the second.
- 0 if both arguments have the same order.
- 1 if the first argument is higher in order compared to the second.

The ordering of the special values and finite numbers is as follows:

```
-NAN<-SNAN<-INFINITY<-0.10<-0.100<-0<0<0.100<0.10<INFINITY<SNAN<NAN
```

The result of the function is a SMALLINT value.

The result can be null; if any argument is null, the result is the null value.

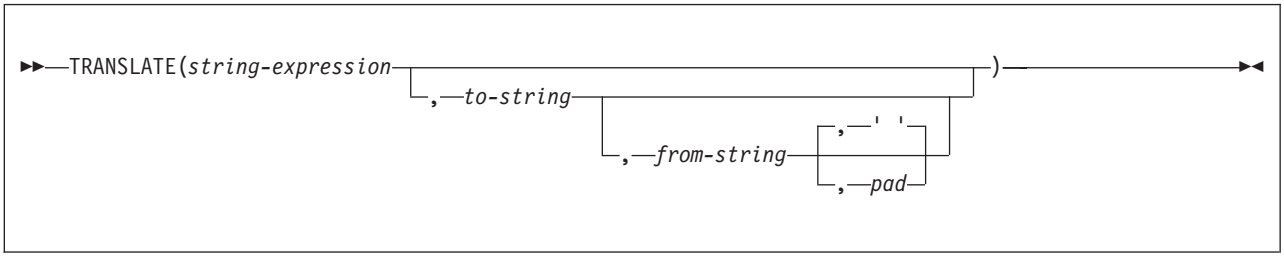
Examples: The following examples show the use of the TOTALORDER function to compare decimal floating point values:

```
TOTALORDER(-INFINITY, -INFINITY)              = 0
TOTALORDER(DECFLOAT(-1.0), DECFLOAT(-1.0))    = 0
TOTALORDER(DECFLOAT(-1.0), DECFLOAT(-1.00))   = -1
TOTALORDER(DECFLOAT(-1.0), DECFLOAT(-0.5))    = -1
TOTALORDER(DECFLOAT(-1.0), DECFLOAT(0.5))     = -1
TOTALORDER(DECFLOAT(-1.0), INFINITY)          = -1
TOTALORDER(DECFLOAT(-1.0), SNAN)              = -1
TOTALORDER(DECFLOAT(-1.0), NAN)               = -1
TOTALORDER(NAN, DECFLOAT(-1.0))               = 1
```

```
TOTALORDER(-NAN, -NAN)                    = 0
TOTALORDER(-SNAN, -SNAN)                  = 0
TOTALORDER(NAN, NAN)                      = 0
TOTALORDER(SNAN, SNAN)                    = 0
```

TRANSLATE

The TRANSLATE function returns a value in which one or more characters of the first argument might have been converted to other characters.



The schema is SYSIBM.

string-expression
An expression that specifies the string to be converted. *string-expression* must return a value that is a built-in character or graphic string data type that is not a LOB. If *string-expression* is an EBCDIC or ASCII graphic string and *string-expression* is the only argument that is specified, the locale name that is specified by the CURRENT LOCALE LC_CTYPE special register must be a non-blank string.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.

to-string
An expression that specifies the characters to which certain characters in *string-expression* are to be converted. This string is sometimes called the *output translation table*. *to-string* must return a value that is a built-in character or graphic string data type that is not a LOB.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.

If the length of *to-string* is less than the length of *from-string*, *to-string* is padded to the length of *from-string* with the *pad* or a blank. If the length of *to-string* is greater than *from-string*, the extra characters in *to-string* are ignored without warning.

from-string
An expression that specifies the characters that if found in *string-expression* are to be converted. This string is sometimes called the *input translation table*. When a character in *from-string* is found, the character in *string-expression* is converted to the character in *to-string* that is in the corresponding position of the character in *from-string*.

from-string must return a value that is a built-in character or graphic string data type that is not a LOB.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.

If *from-string* contains duplicate characters, the first occurrence of the character is used, and no warning is issued. The default value for *from-string* is a string that starts with the character X'00' and ends with the character X'FF' (decimal 255).

pad

An expression that specifies the character with which to pad *to-string* if its length is less than *from-string*. *pad* is an expression that must return a value that is a built-in character or graphic string data type that is not a LOB and has a length of 1. A length of 1 is one single byte for character strings and one double byte string for graphic strings. The default is a blank that is appropriate for *string-expression*.

If string-expression is the only argument that is specified, the string is converted to uppercase based on the locale name that is specified by the CURRENT LOCALE LC_CTYPE special register, as follows:

- blank — SBCS uppercase characters A-Z are converted to SBCS lowercase characters a-z and characters with diacritical marks are not converted. If the string contains MIXED or DBCS characters, full-width Latin uppercase characters A-Z are converted to full-width lowercase characters a-z. For optimal performance, specify a blank string unless your data must be processed by using rules that are defined by a specific locale.
- UNI — The conversion uses both the NORMAL and SPECIAL casing capabilities as described in *z/OS Support for Unicode: Using Unicode Services*. UNI must not be in effect when string-expression is EBCDIC data.
- locale name — The locale defines the rules for conversion to lowercase characters.

For Unicode data, usage of the TRANSLATE function (the TRANSLATE function with one argument is equivalent to the UPPER function) can result in expansion if certain characters are processed. You should ensure that the result string is large enough to contain the result of the expression.

If more than one argument is specified, the result string is built character-by-character from *string-expression* with each character in *from-string* being converted to the corresponding character in *to-string*. For each character in *string-expression*, the *from-string* is searched for the same character. If the character is found to be the *n*th character in *from-string*, the resulting string will contain the *n*th character from *to-string*. If *to-string* is less than *n* characters long, the resulting string will contain the *pad*. If the character is not found in *from-string*, it is moved to the result string without being converted.

The string can contain mixed data. If only one argument is specified, the UPPER function is performed on the argument, and the rules for operating on mixed data in the UPPER function are observed. Full-width Latin lowercase a-z are converted to full-width Latin uppercase letters A-Z. Otherwise, the function operates on a strict byte-count basis, and the result is not necessarily a properly formed mixed data character string.

The encoding scheme of the result is the same as *string-expression*. The data type of the result of the function depends on the data type of *string-expression*, *to-string*, *from-string*, and *pad*:

- VARCHAR if *string-expression* is a character string. The CCSID of the result depends on the arguments:
  - If *string-expression*, *to-string*, *from-string*, or *pad* is bit data, the result is bit data.
  - If *string-expression*, *to-string*, *from-string*, and *pad* are all SBCS:
    - If *string-expression*, *to-string*, *from-string*, and *pad* are all SBCS Unicode data, the CCSID of the result is the CCSID for SBCS Unicode data.

- If *string-expression* is SBCS Unicode data, and *to-string*, *from-string*, or *pad* are not SBCS Unicode data, the CCSID of the result is the mixed CCSID for Unicode data.
- Otherwise, the CCSID of the result is the same as the CCSID of *string-expression*.

– Otherwise, the CCSID of the result is the mixed CCSID that corresponds to the CCSID of *string-expression*. However, if the input is EBCDIC or ASCII and there is no corresponding system CCSID for mixed, the CCSID of the result is the CCSID of *string-expression*.

• VARGRAPHIC if *string-expression* is a graphic. The CCSID of the result is the same as the CCSID of *source-string*.

The result can be null; if the first argument is null, the result is the null value.

Example 1: Return the string 'abcdef' in uppercase characters. Assume that the locale in effect is blank.

```
SELECT TRANSLATE ('abcdef')
  FROM SYSIBM.SYSDUMMY1
```

The result is the value 'ABCDEF'.

Example 2: Assume that host variable *SITE* has a data type of VARCHAR(30) and contains 'Hanauma Bay'.

```
SELECT TRANSLATE (:SITE)
  FROM SYSIBM.SYSDUMMY1
```

Returns the value 'HANAUMA BAY'. The result is all uppercase characters because only one argument is specified.

```
SELECT TRANSLATE (:SITE, 'j', 'B')
  FROM SYSIBM.SYSDUMMY1
```

Returns the value 'Hanauma jay'.

```
SELECT TRANSLATE (:SITE, 'ei', 'aa')
  FROM SYSIBM.SYSDUMMY1
```

Returns the value 'Heneume Bey'.

```
SELECT TRANSLATE (:SITE, 'bA', 'Bay', '%')
  FROM SYSIBM.SYSDUMMY1
```

Returns the value 'HAnAumA bA%'.

```
SELECT TRANSLATE (:SITE, 'r', 'Bu')
  FROM SYSIBM.SYSDUMMY1
```

Returns the value 'Hana ma ray'.

Example 3: Assume that host variable *SITE* has a data type of VARCHAR(30) and contains 'Pivabiska Lake Place'.

```
SELECT TRANSLATE (:SITE, '$$', 'Ll')
  FROM SYSIBM.SYSDUMMY1
```

Returns the value 'Pivabiska $ake P$ace'.

```
SELECT TRANSLATE (:SITE, 'pLA', 'Place', '.')
  FROM SYSIBM.SYSDUMMY1
```

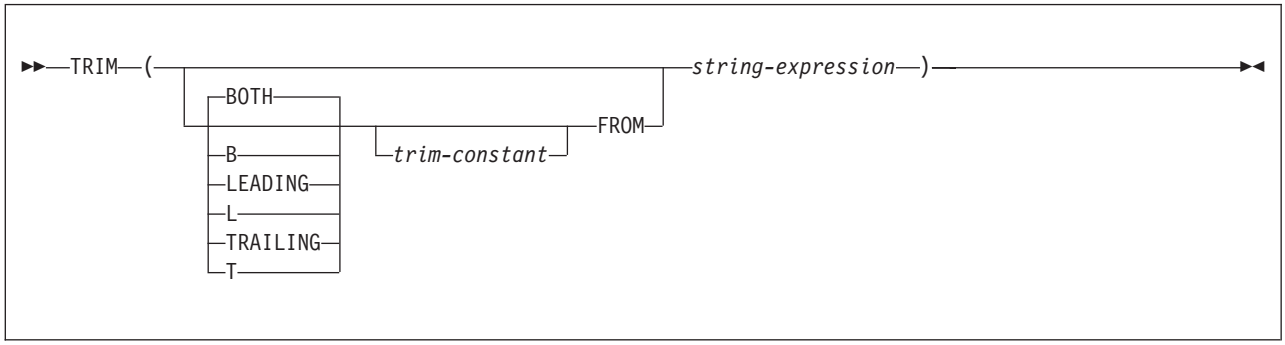
Returns the value 'pivAbiskA LAk. pLA..'.

Related concepts:

z/OS: Unicode Services User's Guide and Reference

TRIM

The TRIM function removes bytes from the beginning, from the end, or from both the beginning and end of a string expression.



The schema is SYSIBM.

The first argument, if specified, indicates whether characters are removed from the end or the beginning of the string. If the first argument is not specified, the characters are removed from both the end and the beginning of the string.

trim-constant
Specifies a constant that indicates the binary, SBCS, or DBCS character that is to be removed. If *string-expression* is a character string, *trim-constant* must be an SBCS or DBCS single-character (2 bytes) constant. If *string-expression* is a binary string, *trim-constant* must be a single-byte binary string constant. If *string-expression* is a DBCS graphic or DBCS-only string, *trim-constant* must be a graphic constant that consists of a single DBCS character.

The default for t*rim-constant* depends on the data type of *string-expression*:

- A DBCS blank if *string-expression* is a DBCS graphic string. For ASCII, the CCSID determines the hex value that represents a DBCS blank. For example, for Japanese (CCSID 301), X'8140' represents a DBCS blank, while for Simplified Chinese, X'A1A1' represents a DBCS blank. For EBCDIC, X'4040' represents a DBCS blank.
- A UTF-16 or UCS-2 blank (X'0020') if *string-expression* is a Unicode graphic string.
- A value of X'00' if *string-expression* is a binary string.
- Otherwise, a single byte blank. For EBCDIC, X'40' represents a blank. When not EBCDIC, X'20' represents a blank.

string-expression
An expression that returns a value that is a built-in character string data type, graphic data type, binary string data type, or numeric data type. *string-expression* must not be a LOB. If *string-expression* is numeric, it is cast to a character string before the function is evaluated. For more information about converting numeric to a character string, see "VARCHAR" on page 673.

string-expression and *trim-expression* must have compatible data types.

The data type of the result depends on the data type of *string-expression*:

- If *string-expression* is a character string data type, the result is VARCHAR. If *string-expression* is defined as FOR BIT DATA the result is FOR BIT DATA.

- If *string-expression* is a graphic string data type, the result is VARGRAPHIC.
- If *string-expression* is a binary string data type, the result is VARBINARY.

The length attribute of the result is the same as the length attribute of *string-expression*. The actual length of the result is the length of *string-expression* minus the number of characters removed. If all of the characters are removed, the result is an empty string.

If *string-expression* can be null, the result can be null; if *string-expression* is null, the result is the null value.

The CCSID of the result is the same as that of *string-expression*.

Example: **Assume the host variable** *HELLO* **of type CHAR(9) has a value of ' Hello '.**

```
SELECT TRIM(:HELLO), TRIM(TRAILING FROM :HELLO)
  FROM SYSIBM.SYSDUMMY1
```

Results in 'Hello' and ' Hello' respectively.

Example: **Assume the host variable** *BALANCE* **of type CHAR(9) has a value of '000345.50'.**

```
SELECT TRIM(L '0' FROM :BALANCE)
  FROM SYSIBM.SYSDUMMY1
```

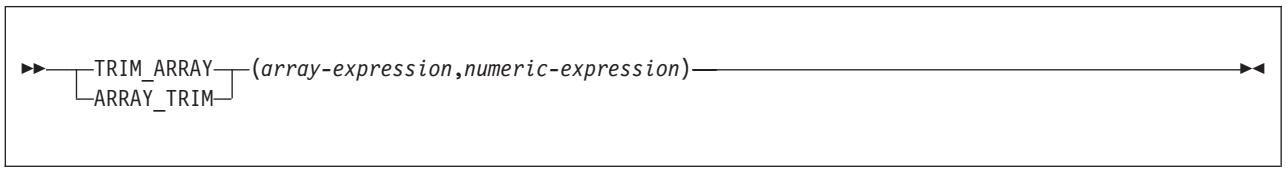
Results in '345.50'

Related reference:

"STRIP" on page 617

TRIM_ARRAY

The TRIM_ARRAY function deletes elements from the end of an ordinary array.



The schema is SYSIBM.

array-expression
An SQL variable or SQL parameter of an array type, or a CAST specification of a parameter marker to an array type. An associative array type cannot be specified.

numeric-expression
Specifies the number of elements that are trimmed from the end of the array. *numeric-expression* can be any numeric data type with a value that can be cast to INTEGER. The value of *numeric-expression* must be greater than or equal to 0 and less than or equal to the cardinality of *array-expression*.

TRIM_ARRAY returns a value with the same array type as *array-expression*, with the cardinality reduced by the value of INTEGER(*numeric-expression*).

The result can be null; if any argument is null, the result is the null value.

The TRIM_ARRAY function can be invoked only in the following contexts:

- A source value for SET *assignment-statement* or SQL PL *assignment-statement*, or a VALUES INTO statement
- The value that is returned in a RETURN statement in an SQL scalar function

Notes

Syntax alternatives: CAST (*SQL-variable* AS *array-type*) can be specified as an alternative to *SQL-variable*. CAST (*SQL-parameter* AS *array-type*) can be specified as an alternative to *SQL-parameter*.

Example 1: Suppose that PHONENUMBERS is a user-defined array type that is defined as an ordinary array. RECENT_CALLS is an array variable of the PHONENUMBERS type. The following statement removes the last element from the array variable RECENT_CALLS.

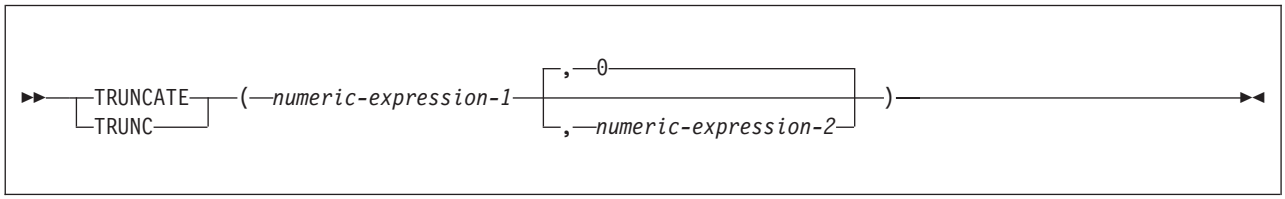
```
SET RECENT_CALLS = TRIM_ARRAY(RECENT_CALLS,1);
```

Example 2: Suppose that INTARRAY is a user-defined array type that is defined as an ordinary array with integer elements. SPECIALNUMBERS and LOWPRIMES are array variables of the INTARRAY type. The SPECIALNUMBERS array contains the values of all the prime numbers less than 1000, which is 168 values. The following statement assigns the 10 smallest prime numbers in the SPECIALNUMBERS array to the first 10 elements of the LOWPRIMES array.

```
SET LOWPRIMES = TRIM_ARRAY(SPECIALNUMBERS,CARDINALITY(SPECIALNUMBERS-10));
```

TRUNCATE or TRUNC

The TRUNCATE function returns the first argument, truncated as specified. Truncation is to the number of places to the right or left of the decimal point this is specified by the second argument.



The schema is SYSIBM.

numeric-expression-1
An expression that returns a value of any built-in numeric data type.

If *expression-1* is a decimal floating-point data type, the DECFLOAT ROUNDING MODE will not be used. The rounding behavior of TRUNCATE corresponds to a value of ROUND_DOWN. If you want a different rounding behavior, use the QUANTIZE function.

The argument can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34).

numeric-expression-2
An expression that returns a value that is a built-in SMALLINT or INTEGER data type. The absolute value of the integer specifies the number of places to truncate. The value of *numeric-expression-2* determines whether truncation is to the right or left of the decimal point.

If *numeric-expression-2* is not negative, *numeric-expression-1* is truncated to the absolute value of *numeric-expression-2* places to the right of the decimal point.

If *numeric-expression-2* is negative, *numeric-expression-1* is truncated to 1 + (the absolute value of *numeric-expression-2*) places to the left of the decimal point. If 1 + (the absolute value of *numeric-expression-2*) is greater than or equal to the number of digits to the left of the decimal point, the result is 0. For example, `TRUNCATE(748.58,-4)` returns 0.

The argument can also be a character string or graphic string data type. The string input is implicitly cast to a numeric value of DECFLOAT(34), which is then assigned to an INTEGER value.

The result of the function has the same data type and length attribute as the first argument.

The result can be null; if any argument is null, the result is the null value.

Example 1: Using sample employee table DSN8B10.EMP, calculate the average monthly salary for the highest paid employee. Truncate the result to two places to the right of the decimal point.

```
SELECT TRUNCATE(MAX(SALARY/12),2)
  FROM DSN8B10.EMP;
```

Because the highest paid employee in the sample employee table earns $52750.00 per year, the example returns the value 4395.83.

Example 2: Return the number 873.726 truncated to 2, 1, 0, -1, -2, -3, and -4 decimal places respectively.

```
SELECT TRUNC(873.726,2),
       TRUNC(873.726,1),
       TRUNC(873.726,0),
       TRUNC(873.726,-1),
       TRUNC(873.726,-2),
       TRUNC(873.726,-3),
       TRUNC(873.726,-4)
  FROM TABLEX
  WHERE INTCOL = 1234;
```

This example returns the values 873.720, 873.700, 873.000, 870.000, 800.000, 0000.000, and 0000.000.

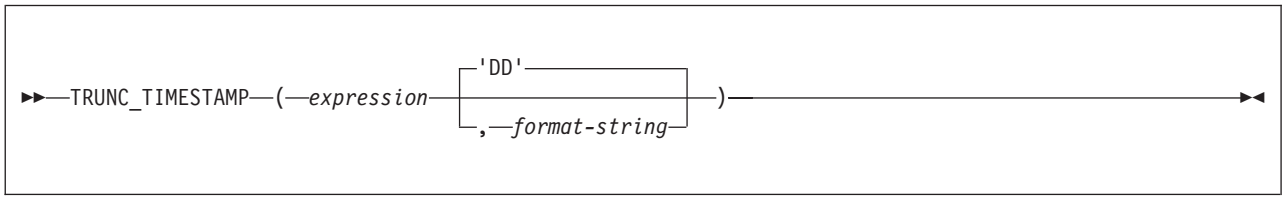
Example 3: Calculate both positive and negative numbers.

```
SELECT TRUNCATE( 3.5, 0),
 TRUNCATE( 3.1, 0),
 TRUNCATE(-3.1, 0),
 TRUNCATE(-3.5, 0)
 FROM TABLEX;
```

This example returns: the values 3.0, 3.0, -3.0, -3.0.

TRUNC_TIMESTAMP

The TRUNC_TIMESTAMP function returns a TIMESTAMP WITHOUT TIME ZONE value that is the *expression*, truncated to the unit that is specified by the *format-string*.



The schema is SYSIBM.

expression
An expression that returns a value of any of the following built-in data types: a timestamp, a character string, or a graphic string. If expression is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a timestamp with an actual length that is not greater than 255 bytes. A time zone in a string representation of a timestamp is ignored. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 101.

format-string
An expression that returns a built-in character string or graphic string data type, with a length that is not greater than 255 bytes. *format-string* contains a template of how the timestamp represented by expression should be truncated. For example, if *format-string* is 'DD', the timestamp that is represented by *expression* is truncated to the nearest day. *format-string* must be a valid template for a timestamp, and not include leading or trailing blanks. Allowable values for *format-string* are listed in the following table.

Table 77. ROUND_TIMESTAMP and TRUNC_TIMESTAMP format models

Format model	Rounding or truncating unit	ROUND_TIMESTAMP example	TRUNC_TIMESTAMP example
CC SCC	Century Rounds up to the start of the next century after the 50th year of the century (for example on 1951–01–01–00.00.00). Not valid for a TIME argument.	Input Value: 1897-12-04-12.22.22.000000 Result: 1901-01-01-00.00.00.000000	Input Value: 1897-12-04-12.22.22.000000 Result: 1801-01-01-00.00.00.000000
SYYYY YYYY YEAR SYEAR YYY YY Y	Year (Rounds up on July 1st)	Input Value: 1897-12-04-12.22.22.000000 Result: 1898-01-01-00.00.00.000000	Input Value: 1897-12-04-12.22.22.000000 Result: 1897-01-01-00.00.00.000000

Table 77. ROUND_TIMESTAMP and TRUNC_TIMESTAMP format models (continued)

Format model	Rounding or truncating unit	ROUND_TIMESTAMP example	TRUNC_TIMESTAMP example
IYYY IYY IY I	ISO Year (Rounds up on July 1st)	Input Value: 1897-12-04-12.22.22.000000 Result: 1898-01-03-00.00.00.000000	Input Value: 1897-12-04-12.22.22.000000 Result: 1897-01-04-00.00.00.000000
Q	Quarter (Rounds up on the sixteenth day of the second month of the quarter)	Input Value: 1999-06-04-12.12.30.000000 Result: 1999-07-01-00.00.00.000000	Input Value: 1999-06-04-12.12.30.000000 Result: 1999-04-01-00.00.00.000000
MONTH MON MM RM	Month (Rounds up on the sixteenth day of the month)	Input Value: 1999-06-18-12.12.30.000000 Result: 1999-07-01-00.00.00.000000	Input Value: 1999-06-18-12.15.00.000000 Result: 1999-06-01-00.00.00.000000
WW	Same day of the week as the first day of the year (Rounds up on the 12th hour of the 3rd day of the week, with respect to the first day of the year)	Input Value: 2000-05-05-12.12.30.000000 Result: 2000-05-06-00.00.00.000000	Input Value: 2000-05-05-12.15.00.000000 Result: 2000-04-29-00.00.00.000000
IW	Same day of the week as the first day of the ISO year (Rounds up on the 12th hour of the 3rd day of the week, with respect to the first day of the ISO year)	Input Value: 2000-05-05-12.12.30.000000 Result: 2000-05-08-00.00.00.000000	Input Value: 2000-05-05-12.15.00.000000 Result: 2000-05-01-00.00.00.000000
W	Same day of the week as the first day of the month (Rounds up on the 12th hour of the 3rd day of the week, with respect to the first day of the month)	Input Value: 2000-05-17-12.12.30.000000 Result: 2000-05-15-00.00.00.000000	Input Value: 2000-05-17-12.15.00.000000 Result: 2000-05-15-00.00.00.000000
DDD DD J	Day (Rounds up on the 12th hour of the day)	Input Value: 2000-05-17-12.59.59.000000 Result: 2000-05-18-00.00.00.000000	Input Value: 2000-05-17-12.59.59.000000 Result: 2000-05-17-00.00.00.000000
DAY DY D	Starting day of the week (Rounds up with respect to the 12th hour of the third day of the week. The first day of the week is always Sunday).	Input Value: 2000-05-17-12.59.59.000000 Result: 2000-05-21-00.00.00.000000	Input Value: 2000-05-17-12.59.59.000000 Result: 2000-05-14-00.00.00.000000
HH HH12 HH24	Hour (Rounds up at 30 minutes)	Input Value: 2000-05-17-23.59.59.000000 Result: 2000-05-18-00.00.00.000000	Input Value: 2000-05-17-23.59.59.000000 Result: 2000-05-17-23.00.00.000000
MI	Minute (Rounds up at 30 seconds)	Input Value: 2000-05-17-23.58.45.000000 Result: 2000-05-17-23.59.00.000000	Input Value: 2000-05-17-23.58.45.000000 Result: 2000-05-17-23.58.00.000000
SS	Second (Rounds up at 500000 microseconds)	Input Value: 2000-05-17-23.58.45.500000 Result: 2000-05-17-23.58.46.000000	Input Value: 2000-05-17-23.58.45.500000 Result: 2000-05-17-23.58.45.000000

If expression is not a TIMESTAMP WITH TIME ZONE value, *expression* is cast as follows:

- If *expression* is a TIMESTAMP WITH TIME ZONE value, *expression* is cast to TIMESTAMP WITHOUT TIME ZONE, with the same precision as *expression*.
- Otherwise, *expression* is cast to TIMESTAMP(6) WITHOUT TIME ZONE.

The result of the function is a timestamp.

The result can be null; if any argument is null, the result is the null value.

The result CCSID is the appropriate CCSID of the argument encoding scheme and the result subtype is the appropriate subtype of the CCSID.

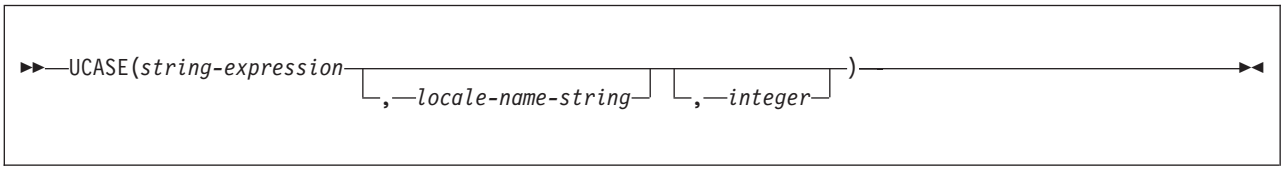
Example: Set the host variable *TRNK_TMSTMP* with the specified date rounded to the nearest year value.

```
SET :TRNK_TMSTMP = TRUNC_TIMESTAMP('2008-03-14-17.30.00', 'YEAR');
```

The host variable *TRNK_TMSTMP* is set with the value '2008-01-01-00.00.00.000000'.

UCASE

The UCASE function returns a string in which all the characters have been converted to uppercase characters, based on the CCSID of the argument. The UCASE function is identical to the UPPER function.

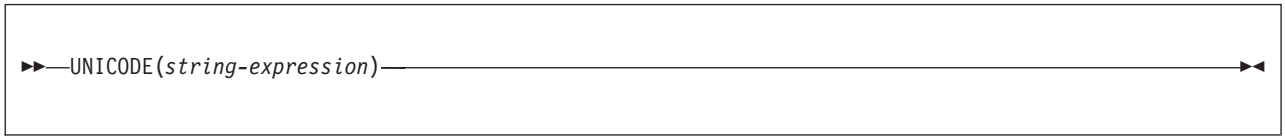


The schema is SYSIBM.

For more information, see "UPPER" on page 668.

UNICODE

The UNICODE function returns the Unicode UTF-16 code value of the leftmost character of the argument as an integer.



The schema is SYSIBM.

string-expression can be of any built-in string data type that is not a LOB.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.

If the argument is ASCII, EBCDIC, or Unicode UTF-8, it is first converted to a Unicode UTF-16 string (CCSID 1200) before the function is executed.

The result of the function is an INTEGER.

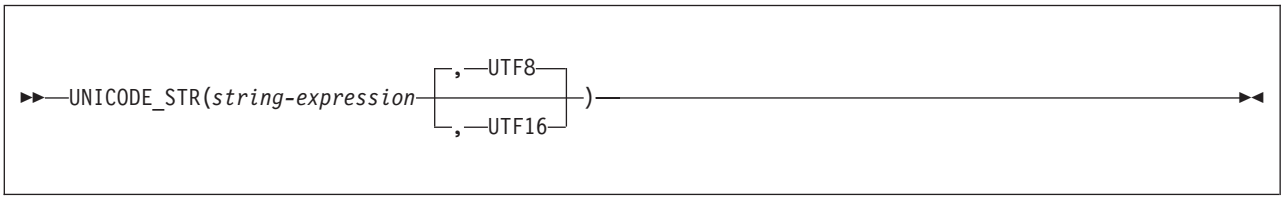
The result can be null; if the argument is null, the result is the null value.

Example: The following example returns the Unicode value of 峰 as an integer and assigns the value to the host variable *hv*:

```
Set :hv = UNICODE('峰');
```
hv is set to an integer with a value '23792'.

UNICODE_STR

The UNICODE_STR function returns a string in Unicode UTF-8 or UTF-16, depending on the specified option. The string represents a Unicode encoding of the input string.



The schema is SYSIBM.

string-expression
An expression that returns a value of a built-in character or graphic string. A character string must not be bit data. Values that are preceded by a backslash ('\') are treated as Unicode UTF-16 characters (for example '\0041' is the Unicode UTF-16 representation for 'A'). A double backslash '\\' indicates a backslash in the string. A partial surrogate character in the expression is replaced with a blank.

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.

UTF8 or UTF16
Specifies the Unicode encoding of the result. If UTF8 is specified, the result is returned as a Unicode UTF-8 character string. If UTF16 is specified, the result is returned as a Unicode UTF-16 graphic string. UTF8 is the default.

The result of the function depends on the second argument:

- VARCHAR if **UTF8** is specified
- VARGRAPHIC if **UTF16** is specified

The length attribute of the result is depends on the second argument (**UTF8** or **UTF16**). The length attribute of the result is calculated using the formulas in Table 30 on page 142. If the result is a character string, the length attribute of the result is `MAX(n,32704)`. If the result is a graphic string, the length attribute of the result is `MAX(n,16352)`. Where *n* is the result of applying the formulas in Table 30 on page 142 based on input and output data types.

If the actual length of the result string exceeds the maximum for the return type, an error occurs.

The result can be null; if the argument is null, the result is the null value.

UNISTR can be specified as a synonym for UNICODE_STR.

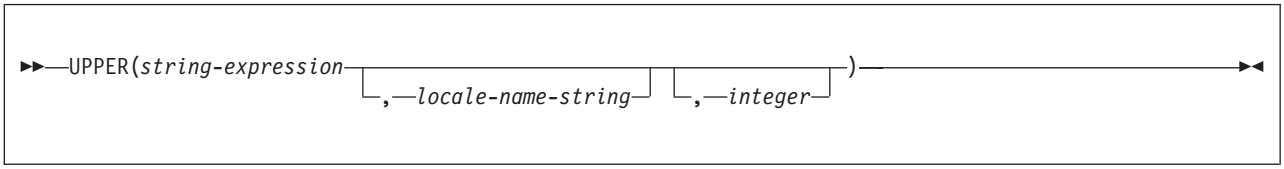
Example: The following example sets the host variable *HV1* to a VARCHAR value that represents the Unicode UTF-8 string that corresponds to the argument:

```
SET :HV1 = UNICODE_STR('Hi, my name is \5CF0');
```

HV1 is assigned a Unicode UTF-8 string with the following value 'Hi, my name is
峰 '

UPPER

The UPPER function returns a string in which all the characters have been converted to uppercase characters.



The schema is SYSIBM.

string-expression
An expression that specifies the string to be converted. *string-expression* must return a value that is a built-in character or graphic string. A character string argument must not be a CLOB, and a graphic string argument must not be a DBCLOB. If *string-expression* is an EBCDIC graphic string, a blank string must not be specified for *locale-name-string*.

locale-name-string
A string constant or a string host variable other than a CLOB or DBCLOB that specifies a valid locale name. If *locale-name-string* is not in EBCDIC, it is converted to EBCDIC. The length of *locale-name-string* must be between 1 and 255 bytes of the EBCDIC representation. The value of *locale-name-string* is not case sensitive and must be a valid locale. For information on locales and their naming conventions, see *z/OS C/C++ Programming Guide*. Some examples of locales include:

Fr_BE
Fr_FR@EURO
En_US
Ja_JP

The conversion process is determined by the value that is specified for the locale name, as follows:

- blank — SBCS uppercase characters A-Z are converted to SBCS lowercase characters a-z, and characters with diacritical marks are not converted. If the string contains MIXED or DBCS characters, full-width Latin uppercase characters A-Z are converted to full-width lowercase characters a-z. For optimal performance, specify a blank string unless your data must be processed by using the rules that are defined by a specific locale.
- UNI — The conversion uses both the NORMAL and SPECIAL casing capabilities as described in *z/OS Support for Unicode: Using Conversion Services*. You must not specify UNI when *string-expression* is EBCDIC data.
- locale name — The locale defines the rules for conversion to lowercase characters.

The value of the host variable must not be null. If the host variable has an associated indicator variable, the value of the indicator variable must not indicate a null value. The locale name must be:

- left justified within the host variable
- padded on the right with blanks if its length is less than that of the host variable and the host variable is in fixed length CHAR or GRAPHIC data type

If *locale-name-string* is not specified, the locale is determined by special register CURRENT LOCALE LC_CTYPE. For information about the special register, see "CURRENT LOCALE LC_CTYPE" on page 177.

If the UPPER function is referenced in an expression-based index, *locale-name-string* must be specified

integer
An integer value that specifies the length attribute of the result. If specified, *integer* must be an integer constant between 1 and 32704 bytes in the representation of the encoding scheme of *string-expression*.

If *integer* is not specified, the length attribute of the result is the same as the length of *string-expression*.

For Unicode data, usage of the UPPER function can result in expansion if certain characters are processed. For example, `UPPER(UX'FB03')` will result in UX'004600460049'. You should ensure that the result string is large enough to contain the result of the expression.

The result can be null; if the argument is null, the result is the null value.

Example 1: Return the string 'abcdef' in uppercase characters. Assume that the locale in effect is blank.

```
SELECT UPPER('abcdef')
  FROM SYSIBM.SYSDUMMY1
```

The result is the value 'ABCDEF'.

Example 2: Return the string 'ffi' in the uppercase characters ('FFI'). Assume that the locale in effect is "UNI".

```
SELECT UPPER(UX'FB03')
  FROM SYSIBM.SYSDUMMYU;
```

This would result in an error because of the expansion that occurs when certain Unicode characters are processed. To avoid the error, you would need to use the following statement instead:

```
SELECT UPPER(CAST(UX'FB03' AS VARCHAR(3))
  FROM SYSIBM.SYSDUMMYU;
```

The result of the preceding statement is the value 'FFI'.

Example 3: Create an index EMPLOYEE_NAME_UPPER for table EMPLOYEE based on built-in function UPPER with locale name 'Fr_FR@EURO'.

```
CREATE INDEX EMPLOYEE_NAME_UPPER
   ON EMPLOYEE (UPPER(LASTNAME, 'Fr_FR@EURO', 60),
                UPPER(FIRSTNAME, 'Fr_FR@EURO', 60),
                ID);
```

The result is the value 'ABCDEF'.

Related concepts:

z/OS: Unicode Services User's Guide and Reference

Related reference:

z/OS XL C/C++ Programming Guide

VALUE

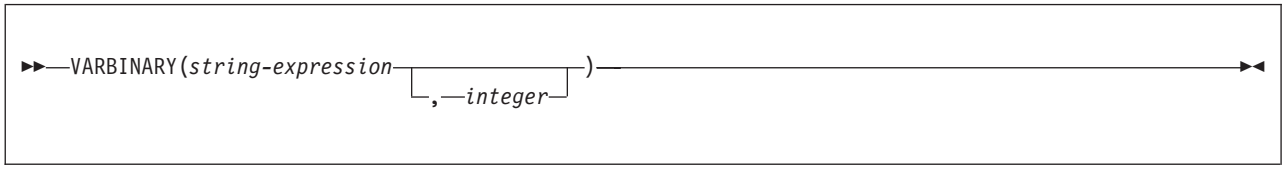
The VALUE function returns the value of the first non-null expression.

The schema is SYSIBM.

Syntax alternatives: The VALUE function can be specified in place of the COALESCE function. COALESCE should be used for conformance to SQL 2003 Core. For more information, see "COALESCE" on page 412.

VARBINARY

The VARBINARY function returns a VARBINARY (varying-length binary string) representation of a string of any type.



The schema is SYSIBM.

string-expression
An expression that returns a value that is a built-in character string, graphic string, binary string, or a row ID type.

integer
An integer value that specifies the length attribute of the resulting binary string. The value must be an integer between 1 and 32704 inclusive. If integer is not specified:

- If the *string-expression* is the empty string constant, the length attribute of the result is 1.
- Otherwise, the length attribute of the result is the same as the length attribute of the *string-expression*, unless the *string-expression* is a graphic string. In this case, the length attribute of the result is twice the length attribute of the *string-expression*.

The result of the function is a varying-length binary string.

The result can be null; if the first argument is null, the result is the null value.

The actual length of the result is the minimum of the length attribute of the result and the actual length of the *string-expression* (or twice the length of the *string-expression* if *string-expression* returns a graphic string). If the length of the *string-expression* is greater than the length attribute of the result, truncation is performed, and a warning is returned unless the *string-expression* is a character string and all the truncated characters are blanks, or the *string-expression* is a graphic string and all the truncated characters are double-byte blanks.

Example 1: The following function returns a varying-length binary string with a length attribute 1, actual length 0, and a value of empty string:

```
SELECT VARBINARY('')
  FROM SYSIBM.SYSDUMMY1;
```

Example 2: The following function returns a varying-length binary string with a length attribute 5, actual length 3, and a value BX'D2C2C8':

```
SELECT VARBINARY('KBH',5)
  FROM SYSIBM.SYSDUMMY1;
```

Example 3: The following function returns a varying-length binary string with a length attribute 3, actual length 3, and a value BX'D2C2C8'

```
SELECT VARBINARY('KBH ',3)
  FROM SYSIBM.SYSDUMMY1;
```

Example 4: The following function returns a varying-length binary string with a length attribute 3, actual length 3, and a value BX'D2C2C8', a warning is also returned.

```
SELECT VARBINARY('KBH-93',3)
  FROM SYSIBM.SYSDUMMY1;
```

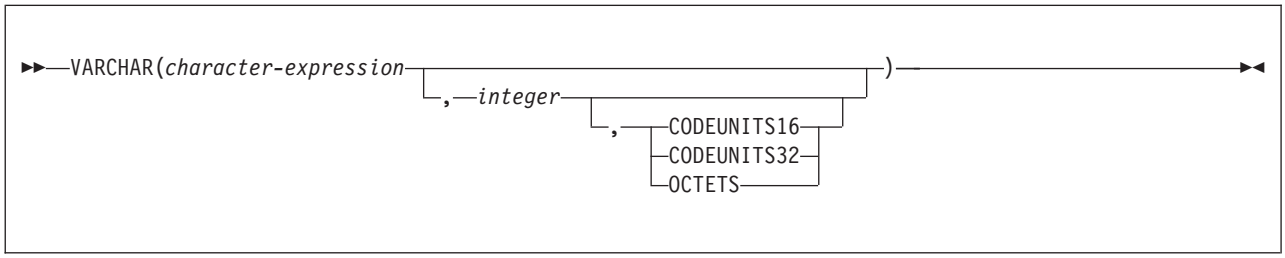
Example 5: The following function returns a varying-length binary string with a length attribute 3, actual length 3, and a value BX'C1C2C3', a warning is also returned.

```
SELECT VARBINARY(BINARY('ABC',5),3)
  FROM SYSIBM.SYSDUMMY1;
```

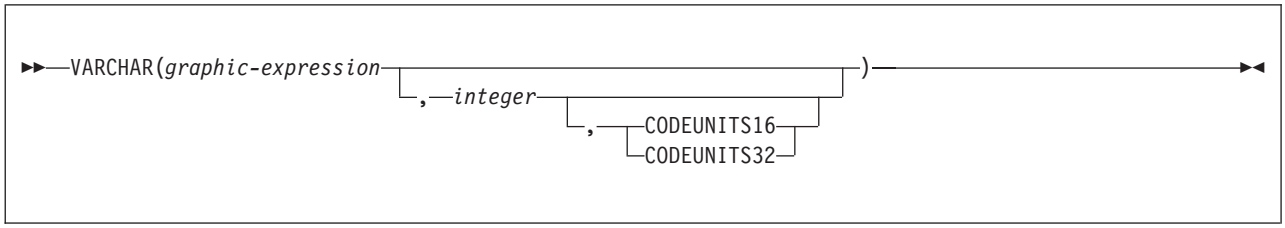
VARCHAR

The VARCHAR function returns a varying-length character string representation of the value specified by the first argument. The first argument can be a character string, a graphic string, a datetime value, an integer number, a decimal number, a floating-point number, or a row ID value.

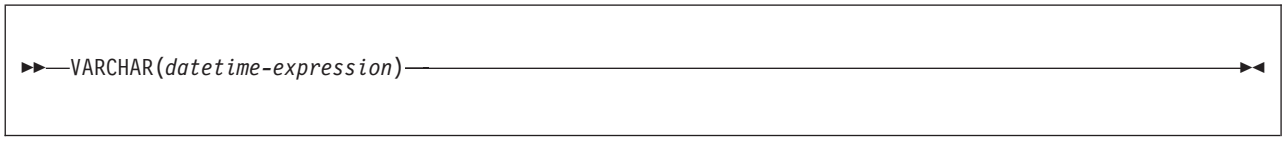
Character to Varchar:



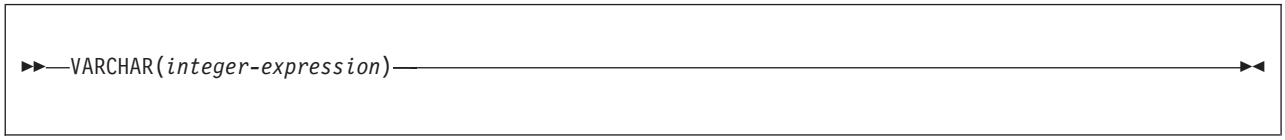
Graphic to Varchar:



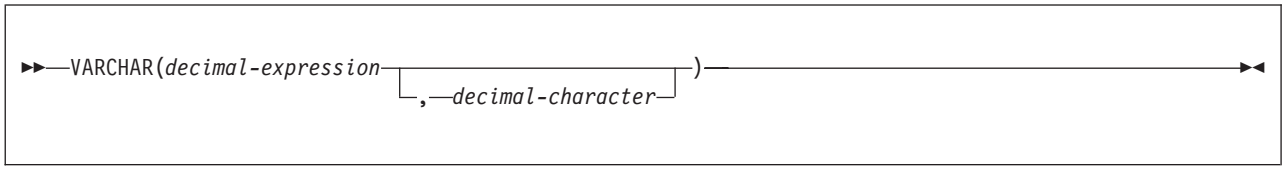
Datetime to Varchar:



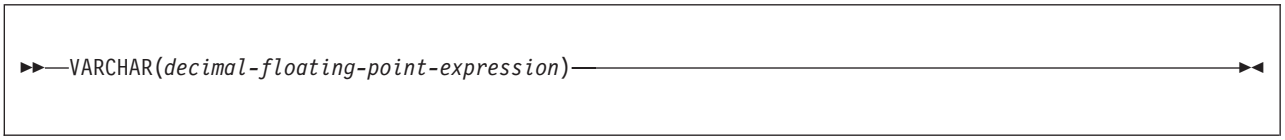
Integer to Varchar:



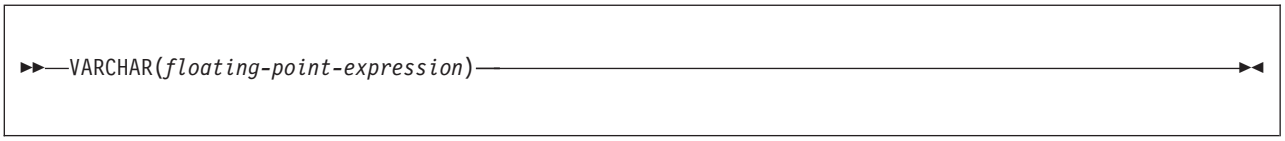
Decimal to Varchar:



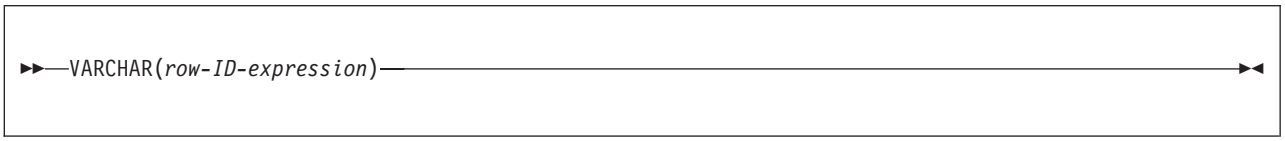
Decimal floating point to Varchar:



Floating-point to Varchar:



Row ID to Varchar:



The schema is SYSIBM.

The result of the function is a varying-length character string (VARCHAR).

The result can be null; if the first argument is null, the result is the null value.

Character to Varchar

character-expression
An expression that returns a value that is a built-in character data type.

integer
Specifies the length attribute for the resulting varying-length character string. The value must be between 1 and 32764, expressed in the units that are either implicitly or explicitly specified. If the length is not specified, the length of the result is the same as the length of *character-expression*.

If CODEUNITS16, CODEUNITS32, or OCTETS is specified, see "Determining the length attribute of the final result" on page 90 for information about how to calculate the length attribute of the result string.

If a length attribute is not specified and if the *character-expression* is an empty string constant, the length attribute of the result is 1 and the result is an empty string. Otherwise, the length attribute of the result is the same as the length attribute of the first argument.

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the unit that is used to express *integer*. If *character-expression* is a character string that is defined as bit data, CODEUNITS16 and CODEUNITS32 cannot be specified.

CODEUNITS16
Specifies that *integer* is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32
Specifies that *integer* is expressed in terms of 32-bit UTF-32 code units.

OCTETS
Specifies that *integer* is expressed in terms of bytes.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String unit specifications" on page 87.

The actual length of the result is the minimum of the length attribute of the result and the actual length of *character-expression*. If the length of *character-expression* is greater than the length attribute of the result, the result is truncated. Unless all the truncated characters are blanks appropriate for *character-expression*, a warning is returned.

If *character-expression* is bit data, the result is bit data. Otherwise, the CCSID of the result is the same as the CCSID of *character-expression*.

Graphic to Varchar

graphic-expression
An expression that returns a value that is a built-in graphic data type.

integer
The length attribute for the resulting varying-length graphic string. The value must be between 1 and 32704, expressed in the units that are either implicitly or explicitly specified.

If CODEUNITS16 or CODEUNITS32 is specified, see "Determining the length attribute of the final result" on page 90 for information about how to calculate the length attribute of the result string.

If a length attribute is not specified, the length attribute of the result is determined as follows (where *n* is the length attribute of the first argument):

- If the *graphic-expression* is the empty graphic string constant, the length attribute of the result is 1.
- If the result is SBCS data, the result length is *n*.
- If the result is mixed data, the result length is `3*(length(graphic-expression)`.

CODEUNITS16 or CODEUNITS32
Specifies the unit that is used to express *integer*.

CODEUNITS16
Specifies that *integer* is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32
Specifies that *integer* is expressed in terms of 32-bit UTF-32 code units.

For more information about CODEUNITS16 or CODEUNITS32, see "String unit specifications" on page 87.

The actual length of the result is the minimum of the length attribute of the result and the actual length of *graphic-expression*. If the length of the graphic expression is greater than the length attribute of the result, the result is truncated. Unless all the truncated characters were blanks appropriate for *graphic-expression*, a warning is returned.

The CCSID of the result is the character mixed CCSID that corresponds to the graphic CCSID of *graphic-expression*.

Datetime to Varchar

datetime-expression
An expression whose value has one of the following three built-in data types:

date The result is a varying-length character string representation of the date in the format that is specified by the DATE precompiler option, if one is provided, or else field DATE FORMAT on installation panel DSNTIP4 specifies the format. If the format is to be LOCAL, field LOCAL DATE LENGTH on installation panel DSNTIP4 specifies the length of the result. Otherwise, the length attribute and actual length of the result is 10.

LOCAL denotes the local format at the DB2 that executes the SQL statement. If LOCAL is used for the format, a time exit routine must be installed at that DB2.

An error occurs if the second argument is specified and is not a valid value.

time The result is a varying-length character string representation of the time in the format specified by the TIME precompiler option, if one is provided, or else field TIME FORMAT on installation panel DSNTIP4 specifies the format. If the format is to be LOCAL, the field LOCAL TIME LENGTH on installation panel DSNTIP4 specifies the length of the result. Otherwise, the length attribute and actual length of the result is 8.

LOCAL denotes the local format at the DB2 that executes the SQL statement. If LOCAL is used for the format, a time exit routine must be installed at that DB2.

An error occurs if the second argument is specified and is not a valid value.

timestamp
The result is the character string representation of the timestamp with time zone. The second argument must not be specified.

- If *datetime-expression* is a TIMESTAMP (0) WITHOUT TIME ZONE, the length of the result is 19.
- If *datetime-expression* is a TIMESTAMP (*p*) WITHOUT TIME ZONE, the length of the result is 20+*p* where p is the timestamp precision. The second argument must not be specified.
- If *datetime-expression* is a TIMESTAMP (0) WITH TIME ZONE, the length of the result is 25.
- If *datetime-expression* is a TIMESTAMP (*p*) WITH TIME ZONE, the length of the result is 26 +*p* where *p* is the timestamp precision. The second argument must not be specified

The CCSID of the result is determined from the context in which the function was invoked. For more information, see "Determining the encoding scheme and CCSID of a string" on page 47.

Integer to Varchar

integer-expression
An expression that returns a value that is a built-in integer data type (SMALLINT, INTEGER, BIGINT).

The result is a varying-length character string representation (VARCHAR) of the argument in the form of an SQL integer constant.

The length attribute of the result depends on whether the argument is a small or large integer as follows:

- If the argument is a small integer, the length attribute of the result is 6 bytes.
- If the argument is a large integer, the length attribute of the result is 11 bytes.
- If the argument is a big integer, the length attribute of the result is 20 bytes.

The actual length of the result is the smallest number of characters that can be used to represent the value of the argument. If the argument is negative, the first character of the result is a minus sign. Otherwise, the first character is a digit.

The CCSID of the result is the SBCS CCSID of the appropriate encoding scheme.

Decimal to Varchar

decimal-expression
An expression that returns a value that is a built-in decimal data type. To specify a different precision and scale for the expression's value, apply the DECIMAL function to the expression before applying the VARCHAR function.

decimal-character
Specifies the single-byte character constant (CHAR or VARCHAR) that is used to delimit the decimal digits in the result character string. The character must not be a digit, a plus sign (+), a minus sign (-), or a blank. The default is the period (.) or comma (,). For information on what factors govern the choice, see "Decimal point representation" on page 328.

The result is a varying-length character string representation of the argument. The result includes a decimal character and up to *p* digits where *p* is the precision of *decimal-expression* with a preceding minus sign if the argument is negative. Leading zeros are not returned. Trailing zeros are returned.

If the function is invoked as VARCHAR9, the result is formatted as described in the preceding paragraph, with the following exceptions:

- The result includes leading zeros from the decimal value.
- A decimal character even if the scale of the decimal value is zero.

The length attribute of the result is 2+*p* where *p* is the precision of *decimal-expression*.

The actual length of the result is the smallest number of characters that can be used to represent the result, except that trailing zeros are included. If the argument is negative, the result begins with a minus sign. Otherwise, the result begins with a digit. If the scale of *decimal-expression* is zero, the decimal character is not returned.[21]

The CCSID of the result is determined from the context in which the function was invoked. For more information, see "Determining the encoding scheme and CCSID of a string" on page 47.

Decimal floating-point to Varchar

decimal-floating-point-expression
An expression that returns a value that is the built-in DECFLOAT data type.

21. If the BIF_COMPATIBILITY system parameter is set to V9_DECIMAL_VARCHAR, or if the SYSCOMPAT_V9.VARCHAR function is used, refer to the *DB2 for z/OS Version 9 SQL Reference* for the result of the VARCHAR function with decimal input.

The result is the varying-length character string representation of the argument in the form of an SQL decimal floating-point constant.

If the DECFLOAT value is one of the special values Infinity, sNaN, or NaN, the strings 'INFINITY', 'SNAN', or 'NAN', respectively, are returned. If the special value is negative, a minus sign is the first character in the returned string. The DECFLOAT special value sNaN does not result in an exception when it is converted to a string.

The length attribute of the result is 42 bytes.

The CCSID of the result is determined from the context in which the function was invoked. For more information, see "Determining the encoding scheme and CCSID of a string" on page 47.

Floating-Point to Varchar

floating-point-expression
An expression that returns a value that is a built-in floating-point data type.

The result is a varying-length character string representation (VARCHAR) of the argument in the form of an SQL floating-point constant.

The length attribute of the result is 24. The actual length of the result is the smallest number of characters that can represent the value of the argument such that the mantissa consists of a single digit other than zero followed by a period and a sequence of digits. If the argument is negative, the first character of the result is a minus sign; otherwise, the first character is a digit. If the argument is zero, the result is '0E0'.

The CCSID of the result is determined from the context in which the function was invoked. For more information, see "Determining the encoding scheme and CCSID of a string" on page 47.

Row ID to Varchar

row-ID-expression
An expression that returns a value that is a built-in row ID data type.

The result is a varying-length character string representation (VARCHAR) of the argument. It is bit data.

The length attribute of the result is 40. The actual length of the result is the length of *row-ID-expression*.

Notes

Syntax alternatives: VARCHAR9 can be specified as an alternative to VARCHAR. The result of the the function is the same, except when the first argument is decimal data. See 'Decimal to Character' for a description of the result.

Examples

Example 1: Assume that host variable *JOB_DESC* is defined as VARCHAR(8). Using sample table DSN8B10.EMP, set *JOB_DESC* to the varying-length string equivalent of the job description (column JOB defined as CHAR(8)) for the employee with the last name of 'QUINTANA'.

```
SELECT VARCHAR(JOB)
  INTO :JOB_DESC
  FROM DSN8B10.EMP
  WHERE LASTNAME = 'QUINTANA';
```

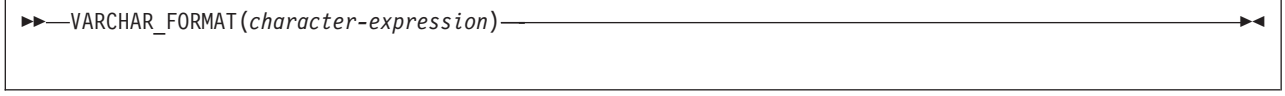
Example 2: FIRSTNME is a VARGRAPHIC(6) column in a Unicode table T1. One of its values is the string 'Jürgen' (X'004A00FC007200670055006E'). When FIRSTNME has this value:

```
Function ...                          Returns ...
-------------------------------------------------------------
VARCHAR(FIRSTNME,3,CODEUNITS32)       'Jür'  -- x'4AC3BC72'
VARCHAR(FIRSTNME,3,CODEUNITS16)       'Jür'  -- x'4AC3BC72'
VARCHAR(FIRSTNME,3,OCTETS)            'Jü'   -- x'4AC3BC'
```

VARCHAR_FORMAT

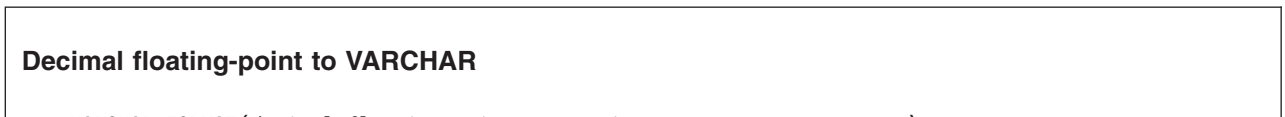
The VARCHAR_FORMAT function returns a character string representation of the first argument in the format indicated by *format-string*.

Character to VARCHAR



Timestamp to VARCHAR

Decimal floating-point to VARCHAR



The schema is SYSIBM.

Character to VARCHAR

character-expression
An expression that returns a value that must be a built-in CHAR or VARCHAR data type. If a supplied argument is a GRAPHIC or VARGRAPHIC data type, it is first converted to VARCHAR before evaluating the function.

The result is a VARCHAR value with a length attribute that matches the length attribute of the argument. The value of the result is the same as the value of *character-expression*.

If *character-expression* returns graphic data, the CCSID of the result is the character mixed CCSID that corresponds to the graphic argument. If *character-expression* returns bit data, the result is bit data. Otherwise, the CCSID of the result is the same as the CCSID of *character-expression*.

Timestamp to VARCHAR

timestamp-expression
An expression that returns a value that must be a DATE or TIMESTAMP, or a valid character string or graphic string representation of a date or timestamp that is not a CLOB or DBCLOB. If the argument is a graphic string representation of a data or timestamp, it is first converted to a character string before evaluating the function.

If *timestamp-expression* is a DATE or a valid string representation of a date, it is first converted to a TIMESTAMP(0) value, assuming a time of exactly midnight (00.00.00). If the HH12 format element is specified and the time component of the first argument is 24:00:00, the input timestamp value is adjusted to 00:00:00 and the date is incremented by one day.

For the valid formats of string representations of datetime values, see "String representations of datetime values" on page 101.

format-string
An expression that returns a built-in character string or graphic string data type that is not a LOB and has a length attribute that is not greater than 255 bytes. If the value is not a CHAR or VARCHAR data type, it is implicitly cast to VARCHAR before the function is evaluated. Leading and trailing blanks are removed from the string. If the argument returns timestamp data type, the resulting substring must conform to the rules for formatting a timestamp. If *expression* returns timestamp with a time zone, the resulting substring must conform to the rules for formatting a timestamp with time zone.

The value is a template for how timestamp-expression is to be formatted.

A valid *format-string* can contain a combination of the format elements listed below. Two format elements can be separated by one or more of the following separator characters.

- minus sign (-)
- period (.)
- forward slash (/)
- comma (,)
- apostrophe (')
- semicolon (;)
- colon (:)
- blank ( )

Separator characters can also be specified at the start or end of *format-string*. *format-string* can also be an empty string, a string of blanks, or a string of separator characters.

The following table lists the valid format elements that *format-string* can contain.

Table 78. Valid format elements of format-string

Format element	Description (assuming the default is to return leading zeros)
AM or PM [1]	Meridian indicator (morning or evening) without periods. This format element uses the exact strings "AM" or "PM".
A.M. or P.M. [1]	Meridian indicator (morning or evening) with periods. This format element uses the exact strings "A.M." or "P.M.".
CC	Century (00-99). If the last two digits of the four digit year are zero, the result is the first two digits of the year. Otherwise, the result is the first two digits of the year plus one.

Table 78. Valid format elements of format-string (continued)

Format element	Description (assuming the default is to return leading zeros)
D [1]	Day of the week (1-7). 1 is Sunday and 7 is Saturday.
DD	Day of the month (01-31).
DDD	Day of the year (001-366).
FF or FF*n*	Fractional seconds (0-999999). The number *n* is used to specify the number of digits to include in the returned value. Valid values for *n* are 1-6 with no leading zeros. Specifying FF is equivalent to specifying FF6. If the timestamp precision of *timestamp-expression* is less than what is specified by the format, zero digits are padded onto the right of the specified digits.
HH	Hour of the day (01-12).
HH12	Hour of the day (01-12).
HH24	Hour of the day (00-24).
I	ISO year (0-9). The last digit of the year based on the ISO week that is returned.
ID	ISO day of the week (1-7). 1 is Monday and 7 is Sunday.
IW	ISO week of the year (01-53). The week starts on Monday and includes 7 days. Week 1 is the first week of the year to contain a Thursday, which is equivalent to the first week of the year to contain January 4.
IY	ISO year (00-99). The last two digits of the year based on the ISO week that is returned.
IYY	ISO year (000-999). The last three digits of the year based on the ISO week that is returned.
IYYY	ISO year (0000-9999). The last four digits of the year based on the ISO week that is returned.
J	Julian date (0000000-9999999).
MI	Minute (00-59).
MM	Month (01-12). January is 01.
MONTH, Month, or month [1, 2]	Name of the month in uppercase, sentence case, or lowercase format in English.
MON, Mon, or mon [1, 2]	Three-character abbreviated name of the month in uppercase, sentence case, or lowercase format in English.

Table 78. Valid format elements of format-string (continued)

Format element	Description (assuming the default is to return leading zeros)
NNNNNN	Microseconds (000000-999999). This format is equivalent to specifying FF6.
Q	Quarter (1-4). January through March is 1.
RRRR	Year (0000-9999). RRRR behaves the same as YYYY.
RR	Last two digits of the year (00-99). RR behaves the same as YY.
SS	Seconds (00-59).
SSSSS	Seconds since the previous midnight (00000-86400).
TZH	Time zone hour. (-24 to +24, This range accommodates daylight saving time changes.)
TZM	Time zone minute (00–59).
W	Week of the month (1-5). Week 1 starts on the first day of the month and ends on the seventh day.
WW	Week of the year (01-53). Week 1 begins on January 1 and ends on January 7.
Y	Last digit of the year (0-9).
YY	Last two digits of the year (00-99).
YYY	Last three digits of the year (000-999).
YYYY	Year (0000-9999).

Notes:

1. This format element is case sensitive. In cases where the format elements are ambiguous, the case insensitive format elements will be considered first.
2. Only these exact spellings and case combinations can be used. If this format element is specified in an invalid case combination an error is returned.

If *expression* is a TIMESTAMP WITHOUT TIME ZONE value, *format-string* must not contain TZH or TZM

The result is a representation of *timestamp-expression* in the format specified by *format-string*. *format-string* is interpreted as a series of format elements that can be separated by one or more separator characters. A string of characters in *format-string* is interpreted as the longest matching format element in the previous table. If two format elements that contain the same characters are not delimited by a separator character, the specification is interpreted, starting from the left, as the longest matching element in the table, and continues until matches are found for the remainder of the format string. For example, 'YYYYYYDD' is interpreted as the format elements, 'YYYY', 'YY', and 'DD'.

If the first argument is timestamp with time zone, or the second argument is a constant that contains a format element for a time zone, the resulting string contains a timestamp with time zone. Otherwise, the resulting string does not contain a time zone.

The result is the varying-length character string that contains *expression* in the format that is specified by *format-string*. The length attribute of the result is the maximum of 255 and the length attribute of *format-string*. The *format-string* determines the actual length of the result. The actual length must not be greater than the length attribute of the result.

The result can be null; if the argument is null, the result is the null value.

The CCSID of the result is determined from the context in which the function is invoked. For more information, see "Determining the encoding scheme and CCSID of a string" on page 47.

Decimal floating-point to VARCHAR

decimal-floating-point-expression
An expression that returns a value of any built-in numeric data type. If the argument is not a decimal floating-point value, it is converted to DECFLOAT(34) for processing.

format-string
An expression that must return a value that is a built-in CHAR, VARCHAR, or numeric data type. If the value is not a CHAR or VARCHAR data type, it is implicitly cast to VARCHAR before evaluating the function. If the supplied argument is a GRAPHIC or VARGRAPHIC data type, it is first converted to VARCHAR before evaluating the function. The actual length must not be greater than 254 bytes.

The value is a template for how *decimal-floating-point-expression* is to be formatted. A *format-string* must contain a valid combination of the listed format elements according to the following rules:

- A sign format element ('S', 'MI', 'PR') can be specified only one time.
- A decimal point format element can be specified only one time.
- Alphabetic format elements must be specified in upper case
- A prefix format element can only be specified at the beginning of the format string, before any format elements that are not prefix format elements. When multiple prefix format elements are specified they can be specified in any order.
- A suffix format element can only be specified at the end of the format string, after any format elements that are not suffix format elements.
- A comma format element must not be the first format element that is not a prefix format element. There can be any number of comma format elements.
- Blanks must not be specified between format elements. Leading and trailing blanks can be specified but are ignored when formatting the result.

Table 79. Format elements for the VARCHAR_FORMAT (decimal floating-point to VARCHAR) function

Format element	Description
0	Represents a digit. Leading zeros in a number are formatted as zeros.
9	Represents a digit that can be included at the specified location. Leading zeros in a number are formatted as blanks.
S	**Prefix** If *decimal-floating-point-expression* is a negative number, a leading minus sign (−) is included at the specified location in the result. If *decimal-floating-point-expression* is a positive number, a leading plus sign (+) is included in the result.
$	**Prefix** A dollar sign ($) is included at the specified location in the result.
MI	**Suffix** If *decimal-floating-point-expression* is a negative number, a trailing minus sign (−) is included in the result. If *decimal-floating-point-expression* is a positive number, a trailing blank is included in the result.
PR	**Suffix** If *decimal-floating-point-expression* is a negative number, a leading less than character (<) and a trailing greater than character (>) are included in the result. If *decimal-floating-point-expression* is a positive number, a leading blank and a trailing blank are included in the result.
, (comma)	Each comma represents a group separator that is included at the specified location in the result provided there would be a character to the left of it that is not a prefix character.
. (period/decimal point)	A period represents the decimal point that is included at the specified location in the result.

If *format-string* is not specified, the function is equivalent to **VARCHAR**(*decimal-floating-point-expression*).

The result is a representation of the *decimal-floating-point-expression* value (which might be rounded) in the format that is specified by *format-string*. Prior to being formatted, the value of *decimal-floating-point-expression* is rounded by using the ROUND function, if the number of digits to the right of the decimal point is less than the number of digit format elements ('0' or '9') to the right of the decimal point in *format-string*. *format-string* is applied according to the following rules:

- The result does not include any digit characters to the left of the decimal point if all of the following conditions are true:
  - -1 < *rounded-input-value* < 1
  - *format-string* does not include a '0' format element to the left of the decimal point
  - *format-string* includes at least one digit format element ('0' or '9') to the right of the decimal point
- The result includes a single 0 character immediately before the implicit or explicit decimal point if all of the following conditions are true:
  - The value of *rounded-input-value* is 0 or -0
  - *format-string* includes only the '9' digit format elements to the left of the implicit or explicit decimal point
  - *format-string* does not include any digit format elements to the right of the decimal point
- If *format-string* includes both '0' and '9' format elements to the left of the decimal point, the position of the first digit format element from the left side of the format string determines the presence of leading blanks or zeroes. All '9' format elements specified after the leftmost '0' format element to the left of the implicit or explicit decimal point are treated the same as if a '0' format element had been specified. For example, the *format-string* value '99099' is the same as the value '99000'.
- If the number of digits to the right of the decimal point in *rounded-input-value* is less than the number of digit format elements to the right of the decimal point in *format-string*, the result includes the number of digit characters to the right of the decimal point that corresponds to the number of digit format elements to the right of the decimal point in *format-string*, padded to the right with zeros.
- If the number of digits to the left of the decimal point in *rounded-input-value* is greater than the number of digit format elements to the left of the decimal point in *format-string*, the result is a string of number sign (#) characters that matches the length that *format-string* produces in the result for valid values.
- If the value of *rounded-input-value* represents any of the positive or negative special values, Infinity, sNaN, or NaN, the string 'INFINITY', 'SNAN', 'NAN', '-INFINITY', '-SNAN', or '-NAN' is returned without using the format that is specified by *format-string*. The decimal floating-point special value sNaN does not result in an exception when converted to a string.
- If format-string does not include any of the sign format elements 'S', ''MI', or 'PR', and the value of *rounded-input-value* is negative, a minus sign (–) is included in the result. Otherwise, a blank is included in the resulting string. The minus sign or blank immediately precedes the first digit of the result to the left of the decimal point, or the decimal point if there are no digits to the left of the decimal point.

The result is a varying-length character string representation of *rounded-input-value*. If a single argument is specified the length attribute is 42. Otherwise the length attribute is 254. The actual length of the result is determined by *format-string*, if specified. Otherwise, the actual length of the result is the smallest number of characters that can represent the value of *rounded-input-value*. If the resulting string exceeds the length attribute of the result, the result will be truncated.

The CCSID of the result is determined from the context in which the function is invoked. For more information, see "Determining the encoding scheme and CCSID of a string" on page 47

Notes

Julian and Gregorian calendar:
For timestamp to a varying length character string, the transition from the Julian calendar to the Gregorian calendar on 15 October 1582 is taken into account by this function.

Determinism:
VARCHAR_FORMAT is a deterministic function.

Using the 'D', 'Y', and 'y' format elements:
DB2 for z/OS does not support the 'DY', 'dy', and 'Dy' format elements that are supported by other platforms. If 'DY' or 'Dy' is specified in the format string, it is interpreted as the 'D' format element followed by the 'Y' or 'y' format element. This behavior might change in a future release. To ensure that a 'D' followed by 'Y' or 'y' is interpreted as two separate format elements, include a separator character after the 'D' format element.

Syntax alternatives:
TO_CHAR can be specified as a synonym for **VARCHAR_FORMAT**.

Example: Timestamp to VARCHAR
Set the character variable *TVAR* to a string representation of the timestamp value of RECEIVED from CORPDATA.IN_TRAY, formatted as 'YYYY-MM-DD HH24:MI:SS.

```
SELECT VARCHAR_FORMAT(RECEIVED,'YYYY-MM-DD HH24:MI:SS')
  INTO :TVAR
  FROM CORPDATA.IN_TRAY;
```

Assuming that the value in the RECEIVED column is 'January 1, 2000 at 10am', the following string is returned:

```
'2000-01-01 10:00:00'
```

Assuming that the value in the RECEIVED column is now one second before the beginning of the year 2000 ('December 31, 1999 at 23:59:59pm', the following string is returned:

```
'1999-12-31 23:59:59'
```

The result would be different if HH12 had been specified instead of HH24 in the format string:

```
'1999-12-31 11:59:59'
```

Example: Timestamp to VARCHAR
Assume that the variable *TMSTAMP* is defined as a TIMESTAMP and has the following value: 2007-03-09-14.07.38.123456. The following examples show several invocations of the function and the resulting string values. The result data type in each case is VARCHAR(255).

```
Function invocation                          Result
-------------------                          ------
VARCHAR_FORMAT(TMSTAMP,'YYYYMMDDHHMISSFF3')  20070309020738123
VARCHAR_FORMAT(TMSTAMP,'YYYYMMDDHH24MISS')   20070309140738
VARCHAR_FORMAT(TMSTAMP,'YYYYMMDDHHMI')       200703090207
VARCHAR_FORMAT(TMSTAMP,'DD/MM/YY')           09/03/07
VARCHAR_FORMAT(TMSTAMP,'MM-DD-YYYY')         03-09-2007
VARCHAR_FORMAT(TMSTAMP,'J')                  2454169
VARCHAR_FORMAT(TMSTAMP,'Q')                  1
VARCHAR_FORMAT(TMSTAMP,'W')                  2
VARCHAR_FORMAT(TMSTAMP,'IW')                 10
VARCHAR_FORMAT(TMSTAMP,'WW')                 10
```

```
VARCHAR_FORMAT(TMSTAMP,'Month')             March
VARCHAR_FORMAT(TMSTAMP,'MONTH')             MARCH
VARCHAR_FORMAT(TMSTAMP,'MON')               MAR
```

Example: Timestamp to VARCHAR

Assume that the variable *DTE* is defined as a DATE and has the value of '2007-03-09'. The following examples show several invocations of the function and the resulting string values. The result data type in each case is VARCHAR(255):

```
 Function invocation                       Result
-------------------                        ------
VARCHAR_FORMAT(DTE,'YYYYMMDD')             20070309
VARCHAR_FORMAT(DTE,'YYYYMMDDHH24MISS')     20070309000000
```

Assuming that today is May 26, 2008, the function returns:

```
26-MAY-2007
```

If the format string is 'YYYY-MON-YYYY', the result would be:

```
2007-MAY-2007
```

Example: Timestamp to VARCHAR

Format the hour of the specified string representation of a timestamp using a 12 hour clock and a 24 hour clock:

```
SELECT
   VARCHAR_FORMAT(TIMESTAMP('1979-04-07-14.00.00.000000'), 'HH'),
   VARCHAR_FORMAT(TIMESTAMP('1979-04-07-14.00.00.000000'), 'HH12'),
   VARCHAR_FORMAT(TIMESTAMP('1979-04-07-14.00.00.000000'), 'HH24'),
   VARCHAR_FORMAT(TIMESTAMP('2000-01-01-00.00.00.000000'), 'HH'),
   VARCHAR_FORMAT(TIMESTAMP('2000-01-01-12.00.00.000000'), 'HH'),
   VARCHAR_FORMAT(TIMESTAMP('2000-01-01-24.00.00.000000'), 'HH'),
   VARCHAR_FORMAT(TIMESTAMP('2000-01-01-00.00.00.000000'), 'HH12'),
   VARCHAR_FORMAT(TIMESTAMP('2000-01-01-12.00.00.000000'), 'HH12'),
   VARCHAR_FORMAT(TIMESTAMP('2000-01-01-24.00.00.000000'), 'HH12'),
   VARCHAR_FORMAT(TIMESTAMP('2000-01-01-00.00.00.000000'), 'HH24'),
   VARCHAR_FORMAT(TIMESTAMP('2000-01-01-12.00.00.000000'), 'HH24'),
   VARCHAR_FORMAT(TIMESTAMP('2000-01-01-24.00.00.000000'), 'HH24')
 FROM SYSIBM.SYSDUMMY1;
```

The previous SELECT statement returns the following values:

```
'02'  '02'  '14'  '12'  '12'  '12'  '12'  '12'  '12'  '00'  '12'  '24'
```

Note that the values '00' and '24' on a 24 hour scale both map to a value of '12' on a 12 hour scale.

Example: Timestamp to VARCHAR

Format the month, day, and hour of the specified string representation of a timestamp using a 24 hour clock, and indicate that the result should not contain leading zeros for the components:

```
SELECT
   VARCHAR_FORMAT(TIMESTAMP('1979-04-07-09.14.00.000000'),
                            'FM MM DD HH24'),
 FROM SYSIBM.SYSDUMMY1;
```

The previous SELECT statement returns the following values:

```
4  7  9
```

Example: Timestamp with time zone to VARCHAR

Assume that column PRSTSZ contains a timestamp with time zone value of '2008-02-29.20.00.000000 -08:00'. The following statement returns the value '2008-02-29 20:00:00.000000 -08:00'.

```
SELECT VARCHAR_FORMAT(PRSTSZ, 'YYYY-MM-DD HH24:MI:SS.NNNNNN TZH:TZM'))
 FROM PROJECT;
```

Example: decimal floating-point to VARCHAR

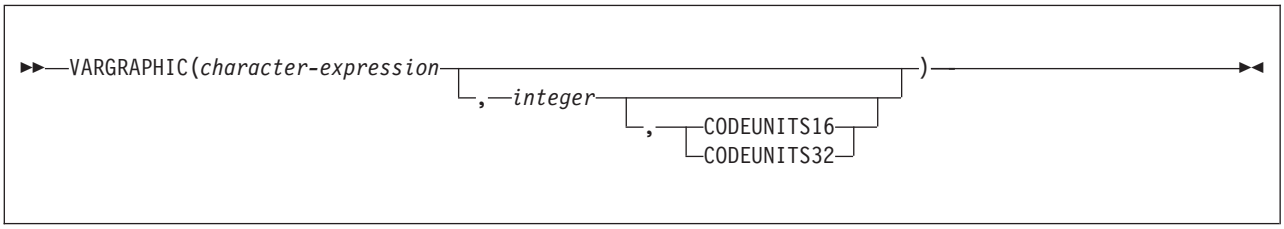
Assume that the variables *POSNUM* and *NEGNUM* are defined as DECFLOAT(34) and have the following values: '1234.56' and '-1234.56', respectively. The following examples show several invocations of the function and the resulting string values. The result data type in each case is VARCHAR(254).

Function invocation	Result
VARCHAR_FORMAT(POSNUM)	'1234.56'
VARCHAR_FORMAT(NEGNUM)	'-1234.56'
VARCHAR_FORMAT(POSNUM,'9999.99')	'1234.56'
VARCHAR_FORMAT(NEGNUM,'9999.99')	'1234.56'
VARCHAR_FORMAT(POSNUM,'99999.99')	' 1234.56'
VARCHAR_FORMAT(NEGNUM,'99999.99')	' 1234.56'
VARCHAR_FORMAT(POSNUM,'00000.00')	'01234.56'
VARCHAR_FORMAT(NEGNUM,'00000.00')	'01234.56'
VARCHAR_FORMAT(POSNUM,'9999.99MI')	'1234.56 '
VARCHAR_FORMAT(NEGNUM,'9999.99MI')	'1234.56-'
VARCHAR_FORMAT(POSNUM,'S9999.99')	'+1234.56'
VARCHAR_FORMAT(NEGNUM,'S9999.99')	'-1234.56'
VARCHAR_FORMAT(POSNUM,'9999.99PR')	' 1234.56 '
VARCHAR_FORMAT(NEGNUM,'9999.99PR')	'<1234.56>'
VARCHAR_FORMAT(POSNUM,'S$9,999.99')	'+$1,234.56'
VARCHAR_FORMAT(NEGNUM,'S$9,999.99')	'-$1,234.56'

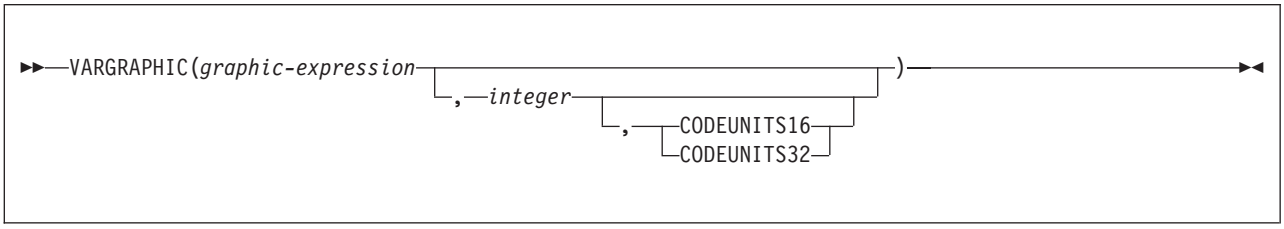
VARGRAPHIC

The VARGRAPHIC function returns a varying-length graphic string representation of a the first argument. The first argument can be a character string value or a graphic string value.

Character to Vargraphic:



Graphic to Vargraphic:



The schema is SYSIBM.

The result of the function is a varying-length graphic string (VARGRAPHIC).

The result can be null; if the first argument is null, the result is the null value.

The length attribute and actual length of the result are measured in double-byte characters because the result is a graphic string.

Character to Vargraphic

character-expression
An expression that returns a value of a built-in character string data type that contains an EBCDIC-encoded or Unicode-encoded character string value. It cannot be BIT data. The argument does not need to be mixed data, but any occurrences of X'0E' and X'0F' in the string must conform to the rules for EBCDIC mixed data. (See "Character strings" on page 84 for these rules.)

integer
The length attribute of the resulting varying-length graphic string. The value must be an integer constant between 1 and 16352.

If CODEUNITS16 or CODEUNITS32 is specified, see "Determining the length attribute of the final result" on page 90 for information about how to calculate the length attribute of the result string.

If *integer* is not specified and if the *character-expression* is an empty string constant or has a value X'0E0F', the length attribute of the result is 1 and the result is an empty string. Otherwise, the length attribute of the result is the same as the length attribute of the first argument.

CODEUNITS16 or CODEUNITS32
Specifies the unit that is used to express *integer*. If CODEUNITS16 or CODEUNITS32 is specified, the input is EBCDIC, and there is no corresponding CCSID for EBCDIC GRAPHIC data, an error occurs.

CODEUNITS16
Specifies that *integer* is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32
Specifies that *integer* is expressed in terms of 32-bit UTF-32 code units.

For more information about CODEUNITS16 and CODEUNITS32, see "String unit specifications" on page 87.

The actual length of the result is the minimum of the length attribute of the result and the actual length of *character-expression*. If the length of *character-expression*, as measured in single-byte characters, is greater than the specified length of the result, as measured in double-byte characters, the result is truncated. Unless all the truncated characters are blanks appropriate for *character-expression*, a warning is returned.

The CCSID of the result is the graphic CCSID that corresponds to the character CCSID of *character-expression*. If the input is EBCDIC and there is no system CCSID for EBCDIC GRAPHIC data, the CCSID of the result is X'FFFE'.

For EBCDIC input data:

Each character of *character-expression* determines a character of the result. The argument might need to be converted to the native form of mixed data before the result is derived. Let *M* denote the system CCSID for mixed data. The argument is not converted if any of the following conditions is true:

- The argument is mixed data and its CCSID is *M*.
- The argument is SBCS data and its CCSID is the same as the system CCSID for SBCS data. In this case, the operation proceeds as if the CCSID of the argument is *M*.

Otherwise, the argument is a new string *S* derived by converting the characters to the coded character set identified by *M*. If there is no system CCSID for mixed data, conversion is to the coded character set that the system CCSID for SBCS data identifies.

The result is derived from *S* using the following steps:

- Each shift character (X'0E' or X'0F') is removed.
- Each double-byte character remains as is.
- Each single-byte character is replaced by a double-byte character.

The replacement for a single-byte character is the equivalent DBCS character if an equivalent exists. Otherwise, the replacement is X'FEFE'. The existence of an equivalent character depends on *M*. If there is no system CCSID for mixed data, the DBCS equivalent of X'*xx*' for EBCDIC is X'42*xx*', except for X'40', whose DBCS equivalent is X'4040'.

For Unicode input data:

Each character of *character-expression* determines a character of the result. The argument might need to be converted to the native form of mixed data before the

result is derived. Let *M* denote the system CCSID for mixed data. The argument is not converted if any of the following conditions is true:

- The argument is mixed data, and its CCSID is *M*.
- The argument is SBCS data, and its CCSID is the same as the system CCSID for SBCS data. In this case, the operation proceeds as if the CCSID of the argument is *M*.

Otherwise, the argument is a new string *S* derived by converting the characters to the coded character set identified by *M*.

The result is derived from *S* using the following steps:

- Each non-supplementary character is replaced by a Unicode double-byte character (a UTF-16 code point). A non-supplementary character in UTF-8 is between 1 and 3 bytes.
- Each supplementary character is replaced by a pair of Unicode double-byte characters (a pair of UTF-16 code points).

The replacement for a single-byte character is the Unicode equivalent character if an equivalent exists. Otherwise, the replacement is X'FFFD'.

Graphic to Vargraphic

graphic-expression
An expression that returns a value of a built-in graphic string data type that contains an EBCDIC-encoded or Unicode-encoded graphic string value.

integer
The length attribute for the resulting varying-length graphic string. The value must be an integer constant between 1 and 16352.

If CODEUNITS16 or CODEUNITS32 is specified, see "Determining the length attribute of the final result" on page 90 for information about how to calculate the length attribute of the result string.

If *integer* is not specified and if the *graphic-expression* is an empty string constant, the length attribute of the result is 1 and the result is an empty string. Otherwise, the length attribute of the result is the same as the length attribute of the first argument.

CODEUNITS16 or CODEUNITS32
Specifies the unit that is used to express *integer*. If CODEUNITS16 or CODEUNITS32 is specified, the input is EBCDIC, and there is no corresponding CCSID for EBCDIC GRAPHIC data, an error occurs.

CODEUNITS16
Specifies that *integer* is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32
Specifies that *integer* is expressed in terms of 32-bit UTF-32 code units.

For more information about CODEUNITS16 and CODEUNITS32, see "String unit specifications" on page 87.

The actual length of the result depends on the number of characters in *graphic-expression*. If the length of *graphic-expression* is greater than the length specified, the result is truncated. Unless all of the truncated characters are double-byte blanks, a warning is returned.

The CCSID of the result is the same as the CCSID of *graphic-expression*.

Example 1: Assume that GRPHCOL is a VARGRAPHIC column in table TABLEX and *MIXEDSTRING* is a character string host variable that contains mixed data. For various rows in TABLEX, an application uses a positioned UPDATE statement to replace the value of GRPHCOL with the value of *MIXEDSTRING*. Before GRPHCOL can be updated, the current value of *MIXEDSTRING* must be converted to a varying-length graphic string. The following statement shows how to code the VARGRAPHIC function within the UPDATE statement to ensure this conversion.

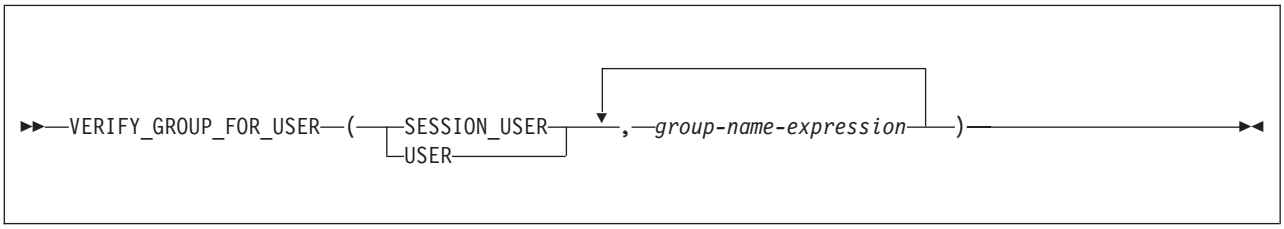
```
EXEC SQL UPDATE TABLEX
  SET GRPHCOL = VARGRAPHIC(:MIXEDSTRING)
  WHERE CURRENT OF CRSNAME;
```

Example 2: FIRSTNAME is a VARCHAR(12) column in table T1. One of its values is the string 'Jürgen'. When FIRSTNAME has this value:

```
Function ...                                Returns ...
-------------------------------------------------------------------
VARGRAPHIC(FIRSTNAME,3,CODEUNITS32)         'Jür'  -- x'004A00FC0072'
VARGRAPHIC(FIRSTNAME,3,CODEUNITS16)         'Jür'  -- x'004A00FC0072'
VARGRAPHIC(FIRSTNAME,3,OCTETS)              An error because OCTETS not allowed
```

VERIFY_GROUP_FOR_USER

The VERIFY_GROUP_FOR_USER function returns a value that indicates whether the primary authorization ID and the secondary authorization IDs that are associated with the first argument are in the authorization names that are specified in the list of the second argument.



The schema is SYSIBM.

SESSION_USER or USER
Specifies the value of the SESSION_USER (or USER) special register.

group-name-expression
An expression that specifies an authorization name. The existence of the authorization name at the current server is not verified. *group-name-expression* must return a built-in character string data type or graphic string data type that is not a LOB. The string must have a length that does not exceed the maximum length of an SQL identifier. The content of the string is not folded to uppercase and is not left justified.

The result of the function is a large integer. The result cannot be null.

The result is 1 if the primary or secondary authorization IDs that are associated with the user that is identified by the SESSION_USER (or USER) special register is in the list that is specified by *group-name-expression*. Otherwise, the result is 0.

The VERIFY_GROUP_FOR_USER function is deterministic within a connection. It is not deterministic across connections. The function can be referenced in a CREATE MASK or a CREATE PERMISSION statement and is considered for table expressions or the merging of views.

Example: In the following example, the EMPLOYEE table has column access control enabled. If the connection is established outside a trusted context and Mary, who has a secondary authorization ID of "MGR", queries the social security number of Tom from the EMPLOYEE table, the social security number is returned. When Mary is no longer a manager, the same query displays the last four digits of Tom's social security number.

Assume that a user who has SECADM authority has created the following column mask:

```
CREATE MASK SSN_MASK ON EMPLOYEE
 FOR COLUMN SSN
 RETURN
 CASE WHEN VERIFY_GROUP_FOR_USER(SESSION_USER, 'MGR') = 1
  THEN SSN
  ELSE 'XXX-XX-' || SUBSTR(SSN, 8, 4)
```

```
 END
 ENABLE;

COMMIT;
```

An ALTER TABLE statement is then issued to activate the column mask on the EMPLOYEE table:

```
ALTER TABLE EMPLOYEE
 ACTIVATE COLUMN ACCESS CONTROL;

COMMIT;
```

Mary connects to DB2, issues the following query, then disconnects from DB2:

```
SELECT SSN
 FROM EMPLOYEE
 WHERE NAME = 'Tom';
```

Mary receives Tom's social security number.

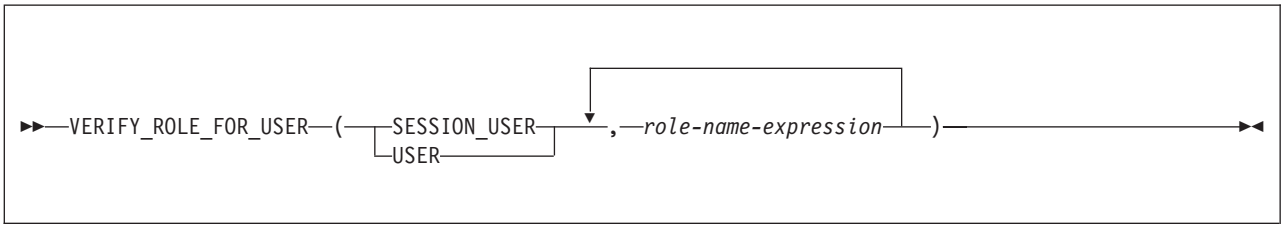
When Mary is no longer a manager, the secondary authorization ID, MGR is removed for her authorization ID. The next time Mary connects to DB2 and issues the following command, only the last four digits of Tom's social security number are displayed because of the column mask SSN_MASK:

```
SELECT SSN
 FROM EMPLOYEE
 WHERE NAME = 'Tom';
```

VERIFY_ROLE_FOR_USER

The VERIFY_ROLE_FOR_USER function returns a value that indicates whether the roles that are associated with the authorization ID that is specified in the first argument are included in the role names that are specified in the list of the second argument.

If the only way to acquire a role is under a trusted connection that is associated with a trusted context, the VERIFY_ROLE_FOR_USER function is equivalent to the VERIFY_TRUSTED_CONTEXT_ROLE_FOR_USER function.



The schema is SYSIBM.

SESSION_USER or USER
Specifies the value of the SESSION_USER (or USER) special register.

role-name-expression
An expression that specifies a role name. The existence of the role name at the current server is not verified. *role-name-expression* must return a built-in character string data type or graphic string data type that is not a LOB. The string must have a length that does not exceed the maximum length of an SQL identifier. The content of the string is not folded to uppercase and is not left justified.

The result of the function is a large integer. The result cannot be null.

The result is 1 if any of the roles that are associated with the user that is identified by the SESSION_USER (or USER) special register is in the list of roles specified by *role-name-expression*. Otherwise, the result is 0.

The VERIFY_ROLE_FOR_USER function is deterministic within a trusted connection. It is not deterministic across trusted connections. The function can be referenced in a CREATE MASK or a CREATE PERMISSION statement and is considered for table expressions or the merging of views.

Example 1: Assume that the following statements have been issued to create specific roles and the trusted context CTX1:

```
CREATE ROLE EMPLOYEE;
COMMIT;

CREATE ROLE MGR;
COMMIT;

CREATE ROLE PAYROLL;
COMMIT;

CREATE TRUSTED CONTEXT CTX1
 BASED UPON CONNECTION USING SYSTEM AUTHID ADMF001
 ATTRIBUTES (ADDRESS '9.30.131.203', ENCRYPTION 'LOW')
 DEFAULT ROLE EMPLOYEE
```

```
 ENABLE
 WITH USE FOR SAM, JOE ROLE MGR WITH AUTHENTICATION;

COMMIT;
```

Joe, who is a manager, issues the following dynamic query through the trusted connection CTX1 to view the salaries of the employees in the DSN8910.EMP table that are in his department:

```
SELECT SALARY FROM DSN8910.EMP
 WHERE VERIFY_ROLE_FOR_USER(SESSION_USER,'MGR','PAYROLL')= 1
 AND WORKDEPT = ?;
```

Example 2: For the following example, suppose that a user with SECADM authority needs to control access for specific users who execute a statement that is accessing a table:

Is the current user, B, using role X to run a statement owned by user C

```
SESSION_USER = B AND
VERIFY_ROLE_FOR_USER(SESSION_USER, 'X')
```

Is the current user, B, using role X to run a statement owned by role D

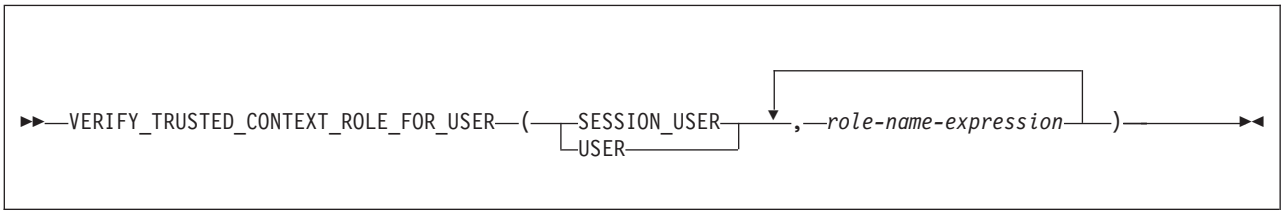
```
SESSION_USER = B AND
VERIFY_ROLE_FOR_USER(SESSION_USER, 'X')
```

Is the current user, B, using role B to execute a dynamic statement

```
SESSION_USER = B AND
VERIFY_ROLE_FOR_USER(SESSION_USER, 'B')
```

VERIFY_TRUSTED_CONTEXT_ROLE_FOR_USER

The VERIFY_TRUSTED_CONTEXT_FOR_USER function returns a value that indicates whether the authorization ID that is associated with first argument has acquired a role in a trusted connection and whether that acquired role is included in the role names that are specified in the list of the second argument.



The schema is SYSIBM.

SESSION_USER or USER
Specifies the value of the SESSION_USER (or USER) special register.

role-name-expression
An expression that specifies a role name. The existence of the role name at the current server is not verified. *role-name-expression* must return a built-in character string data type or graphic string data type that is not a LOB. The string must have a length that does not exceed the maximum length of an SQL identifier. The content of the string is not folded to uppercase and is not left justified.

The result of the function is a large integer. The result cannot be null.

The result is 1 if the user that is identified by the SESSION_USER (or USER) special register has acquired a role under a trusted connection that is associated with a trusted context and that role is in the list of *role-name-expression*. Otherwise, the result is 0.

The VERIFY_TRUSTED_CONTEXT_ROLE_FOR_USER function is deterministic within a trusted connection. It is not deterministic across trusted connections. The function can be referenced in a CREATE MASK or a CREATE PERMISSION statement and is considered for table expressions or the merging of views.

Example 1: Assume that the following statements have been issued to create specific roles and the trusted context CTX1:

```
CREATE ROLE EMPLOYEE;
COMMIT;

CREATE ROLE MGR;
COMMIT;

CREATE ROLE PAYROLL;
COMMIT;

CREATE TRUSTED CONTEXT CTX1
 BASED UPON CONNECTION USING SYSTEM AUTHID ADMF001
 ATTRIBUTES (ADDRESS '9.30.131.203', ENCRYPTION 'LOW')
 DEFAULT ROLE EMPLOYEE
 ENABLE
 WITH USE FOR SAM, JOE ROLE MGR WITH AUTHENTICATION;

COMMIT;
```

Joe, who is a manager, issues the following dynamic query through the trusted connection CTX1 to view the salaries of the employees in the DSN8910.EMP table that are in his department:

```
SELECT SALARY FROM DSN8910.EMP
 WHERE VERIFY_TRUSTED_CONTEXT_ROLE_FOR_USER(SESSION_USER,'MGR','PAYROLL')= 1
 AND WORKDEPT = ?;
```

Example 2: For the following example, suppose that a user with SECADM authority needs to control access for specific users who execute a statement that is accessing a table:

Is the current user, B, using role X to run a statement owned by user C

```
SESSION_USER = B AND
VERIFY_TRUSTED_CONTEXT_ROLE_FOR_USER(SESSION_USER, 'X')
```

Is the current user, B, using role X to run a statement owned by role D

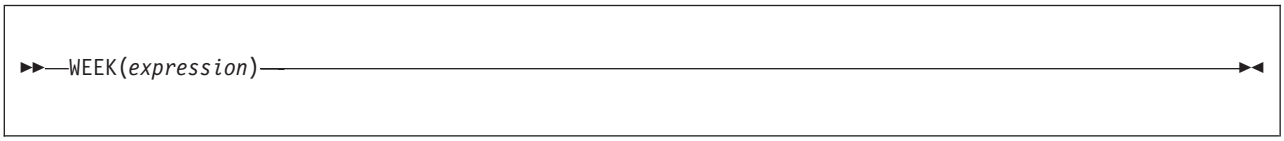
```
SESSION_USER = B AND
VERIFY_TRUSTED_CONTEXT_ROLE_FOR_USER(SESSION_USER, 'X')
```

Is the current user, B, using role B to execute a dynamic statement

```
SESSION_USER = B AND
VERIFY_TRUSTED_CONTEXT_ROLE_FOR_USER(SESSION_USER, 'B')
```

WEEK

The WEEK function returns an integer in the range of 1 to 54 that represents the week of the year. The week starts with Sunday, and January 1 is always in the first week.



The schema is SYSIBM.

The argument must be an expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, or a graphic string. If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 101.

If *expression* is a timestamp with a time zone, or a valid string representation of a timestamp with a time zone, the result is determined from the UTC representation of the datetime value.

The result of the function is a large integer.

The result can be null; if the argument is null, the result is the null value.

Example 1: Using sample table DSN8B10.PROJ, set the integer host variable *WEEK* to the week of the year that project 'AD2100' ended.

```
SELECT WEEK(PRENDATE)
  INTO :WEEK
  FROM DSN8B10.PROJ
  WHERE PROJNO = 'AD2100';
```

The result is that *WEEK* is set 6.

Example 2: The following invocations of the WEEK function returns the same result:

```
SELECT WEEK('1993-08-10-20.00.00'),
   WEEK('1993-08-10-20.00.00-08:00'),
   WEEK('1993-08-10-20.00.00+09:00')
  FROM SYSIBM.SYSDUMMY1;
```

For each invocation of the WEEK function in this SELECT statement, the result is 33.

When the input argument contains a time zone, the result is determined from the UTC representation of the input value. The string representations of a timestamp with a time zone in the SELECT statement all have the same UTC representation: '1993-08-10-20.00.00'.

WEEK_ISO

The WEEK_ISO function returns an integer in the range of 1 to 53 that represents the week of the year. The week starts with Monday and includes seven days. Week 1 is the first week of the year that contains a Thursday, which is equivalent to the first week that contains January 4.

With the WEEK_ISO function, the first one, two, or three days in January might be included in the last week of the previous year. Likewise, the last one, two, or three days in December might be included in the first week of the next year.

The schema is SYSIBM.

The argument must be a date, a timestamp, or a valid string representation of a date or timestamp. A string representation must not be a CLOB or DBCLOB value and must have an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 101.

If *expression* is a timestamp with a time zone, or a valid string representation of a timestamp with a time zone, the result is determined from the UTC representation of the datetime value.

The result of the function is a large integer.

The result can be null; if the argument is null, the result is the null value.

Example 1: Using sample table DSN8B10.PROJ, set the integer host variable *WEEKISO* to the week of the year that project 'AD2100' ended.

```
SELECT WEEK_ISO(PRENDATE)
  INTO :WEEKISO
  FROM DSN8B10.PROJ
  WHERE PROJNO = 'AD2100';
```

Example 2: The following list shows what is returned by the WEEK_ISO function for various dates.

```
DATE:          WEEK_ISO returns:
--------------------------------
2003-12-28      52
2003-12-31       1
2004-01-01       1
2005-01-01      53
2005-01-04       1
2005-12-31      52
2006-01-01      52
2006-01-03       1
```

Example 3: The following invocations of the WEEK_ISO function returns the same result:

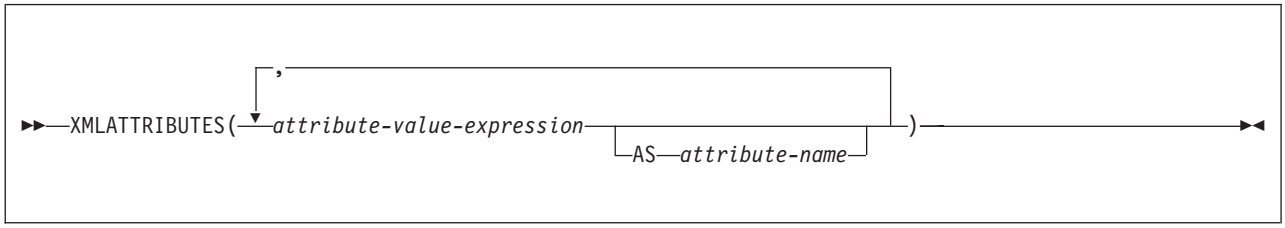
```
SELECT WEEK_ISO('1993-08-10-20.00.00'),
   WEEK_ISO('1993-08-10-20.00.00-08:00'),
   WEEK_ISO('1993-08-10-20.00.00+09:00')
  FROM SYSIBM.SYSDUMMY1;
```

For each invocation of the WEEK_ISO function in this SELECT statement, the result is 32.

When the input argument contains a time zone, the result is determined from the UTC representation of the input value. The string representations of a timestamp with a time zone in the SELECT statement all have the same UTC representation: '1993-08-10-20.00.00' .

XMLATTRIBUTES

The XMLATTRIBUTES function constructs XML attributes from the arguments. This function can be used as an argument only for the XMLELEMENT function.



The schema is SYSIBM.

The result is an XML sequence that contains an XQuery attribute node for each non-null *attribute-value-expression* argument.

attribute-value-expression
An expression that returns a value for the attribute. The data type of *attribute-value-expression* must not be ROWID, a LOB, a distinct type that is based on a ROWID or a LOB, or XML.

The result of *attribute-value-expression* is mapped to an XML value according to the rules for mapping an SQL value to an XML value. If the expression is not a simple column reference, an attribute name must be specified.

AS *attribute-name*
Specifies an attribute name. The name is an SQL identifier that must be in the form of an XML qualified name, or QName. If *attribute-name* is a qualified name, the namespace prefix must be declared within the scope of the qualified name.

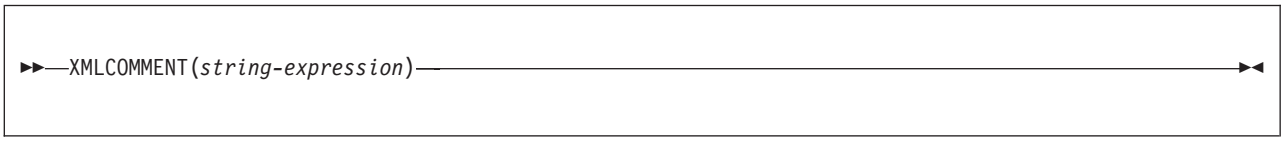
attribute-name cannot be 'xmlns' or prefixed with 'xmlns:'. A namespace is declared using the function XMLNAMESPACES. The attribute names for an element must be unique for the XML element to be well-formed.

If *attribute-name* is not specified, the expression for *attribute-value* must be a column name. The attribute name will be created from the column name using the fully escaped mapping from a column name to an XML attribute name.

The result of the function is an XML value. The result can be null; if all *attribute-value-expression* arguments are null, the result is the null value.

XMLCOMMENT

The XMLCOMMENT function returns an XML value with a single comment node from a string expression. The content of the comment node is the value of the input string expression, mapped to Unicode (UTF-8).



The schema is SYSIBM.

string-expression
An expression that returns a value of a built-in character or graphic string that is not a LOB and is not bit data. The result of *string-expression* is converted to UTF-8 and then parsed to check for conformance to the content of XML comment as specified by the following rules:

- '--' (double-hyphen) must not occur in the string expression
- The string expression must not end with a hyphen ('-')
- Each character of the string can be any Unicode character, excluding the surrogate blocks, X'FFFE', and X'FFFF'

If *string-expression* does not conform to the previous rules, an error is returned.

The result of the function is an XML value that is an XML sequence that contains one XML comment node.

The result can be null; if the argument is null, the result is the null value.

Example: Generate an XML comment:

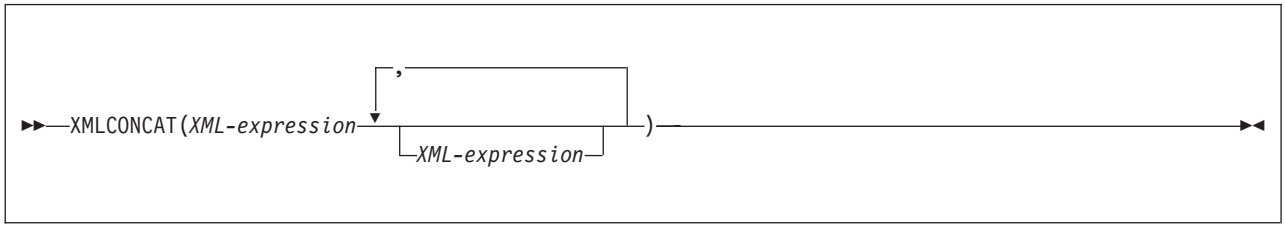
```
SELECT XMLCOMMENT('This is an XML comment')
  FROM SYSIBM.SYSDUMMY1;
```

The result of the query would look similar to the following result:

```
<!--This is an XML comment-->
```

XMLCONCAT

The XMLCONCAT function returns an XML sequence that contains the concatenation of a variable number of XML input arguments.



The schema is SYSIBM.

XML-expression
An expression that returns an XML value.

The data type of the result is XML. The result of the function is an XML sequence that contains the concatenation of the non-null input XML values. Null values in the input are ignored. The result can be null; if the result of every input value is null, the result is the null value.

Example: Concatenate first name and last name elements by using 'first' and 'last' element names for each employee.

```
SELECT XMLSERIALIZE( XMLCONCAT
                ( XMLELEMENT ( NAME "first", e.fname),
                  XMLELEMENT ( NAME "last", e.lname)
                        ) ) AS "result"
FROM employees e;
```

The result of the query would look similar to the following result:

```
result
 --------------------------------------------

 <first>John</first><last>Smith</last>
 <first>Mary</first><last>Smith</last>
```

XMLDOCUMENT

The XMLDOCUMENT function returns an XML value with a single document node and zero or more nodes as its children. The content of the generated XML document node is specified by a list of expressions.

The schema is SYSIBM.

XML-expression
An expression that returns an XML value. A sequence item in the XML value must not be an attribute node. If *XML-expression* returns a null value, it is ignored for further processing. However, if all *XML-expression* values are null, the result of the function is the null value.

The result of the function is an XML value.

The result can be null; if all of the arguments are null, the result is the null value.

The resulting XML value is built from the list of *XML-expression* arguments. The children of the resulting document node are constructed as follows:

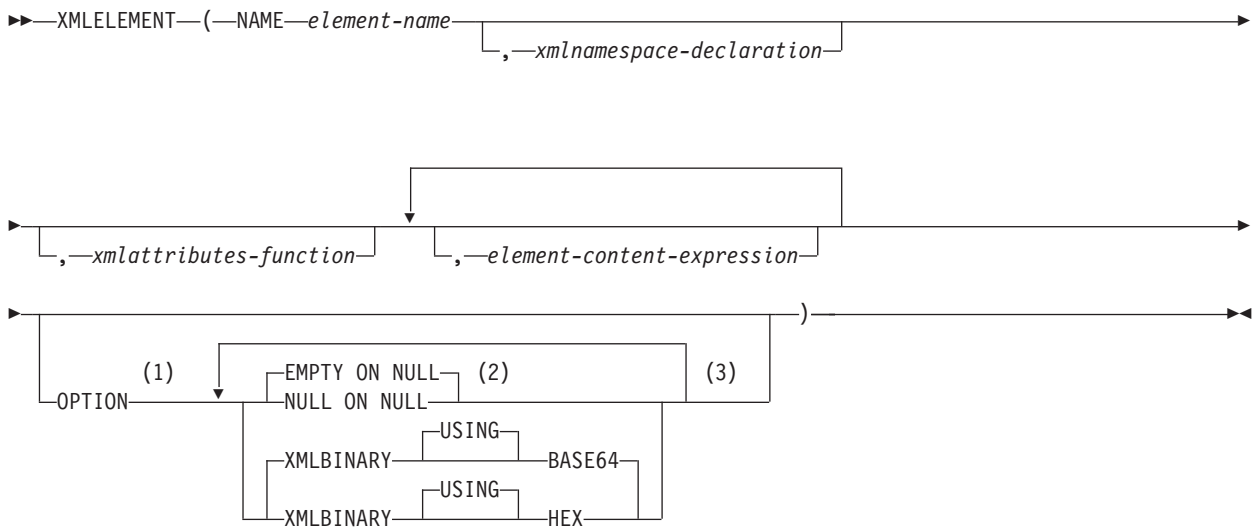
1. All of the non-null XML values that are returned by *XML-expression* are concatenated together. The result is a sequence of nodes or atomic values, which is referred to in the following steps as the *input sequence*. Any document node in the input sequence is replaced by copies of its children.
2. For each node in the input sequence, a new deep copy of the node is constructed. A *deep copy* of a node is a copy of the whole subtree that is rooted at that node, including the node itself and its descendants and attributes. Each copied node has a new node identity. Copied element nodes are given the type annotation 'xdt:untyped', and copied attribute nodes are given the type annotation 'xdt:untypedAtomic'. For each adjacent sequence of one or more atomic values that is returned in the input sequence, a new text node is constructed that contains the result of casting each atomic value to a string, with a single blank character inserted between adjacent values. The resulting sequence of nodes is called the *content sequence*. Adjacent text nodes in the content sequence are merged into a single text node by concatenating the contents of the text nodes with no intervening blanks. After concatenation, any text node that contains a zero-length string is deleted from the content sequence.
3. The nodes in the content sequence become the children of the new document node.

Example 1: Insert a constructed document into an XML column:

```
INSERT INTO T1 VALUES(123,
  (SELECT XMLDOCUMENT(XMLELEMENT(NAME "Emp",
          e.fname || ' ' || e.lname),
          XMLCOMMENT('This is just a simple example'))
    FROM EMPLOYEE e
    WHERE e.empid = 123));
```

XMLELEMENT

The XMLELEMENT function returns an XML value that is an XML element node.



Notes:

1 The OPTION clause can only be specified if at least one *xmlattributes-function* or *element-content-expression* is specified

2 If *element-content-expression* is not specified, EMPTY ON NULL and NULL ON NULL must not be specified.

3 The same clause must not be specified more than one time.

The schema is SYSIBM.

NAME *element-name*
Specifies the name of an XML element. *element-name* is an SQL identifier that must be in the form of an XML qualified name, or QName. If the name is qualified, the namespace prefix must be declared within the scope.

xmlnamespaces-declaration
Specifies the XML namespace declarations that are the result of the XMLNAMESPACES function. The namespaces that are declared are in the scope of the XMLELEMENT function. The namespaces apply to any nested XML functions within the XMLELEMENT function, regardless of whether or not they appear inside another subselect. See "XMLNAMESPACES" on page 718 for more information on declaring XML namespaces.

If *xmlnamespaces-declaration* is not specified, namespace declarations are not associated with the constructed XML element node.

xmlattributes-function
Specifies the attributes for the XML element. The attributes are the result of the XMLATTRIBUTES function. See "XMLATTRIBUTES" on page 703 for more information on constructing attributes.

If *xmlattributes-function* is not specified, attributes are not explicitly part of the constructed XML element node.

element-content-expression
The content of the generated XML element node is specified by an expression or a list of expressions. Each *element-content-expression* must return a value of any built-in data type or distinct type. The expression is used to construct the namespace declarations, attributes, and content of the constructed element node.

If *element-content-expression* is not specified, an empty string is used as the content for the element and NULL ON NULL or EMPTY ON NULL must not be specified.

OPTION
Specifies additional options for constructing the XML element. This clause has no impact on nested invocations of the XMLELEMENT function invocations that are specified in *element-content-expression*.

EMPTY ON NULL or NULL ON NULL
Specifies if a null value or an empty element is returned when the values of each *element-content-expression* is a null value. This option only affects null handling of element contents, not attribute values. The option is not inherited by a nested invocation of XMLELEMENT function within an *element-content-expression*.

EMPTY ON NULL
If the value of each *element-content-expression* is null, an empty element is returned.

EMPTY ON NULL is the default.

NULL ON NULL
If the value of each *element-content-expression* is null, a null value is returned.

XMLBINARY USING BASE64 or XMLBINARY USING HEX
Specifies the assumed encoding of binary input data, character string data with the FOR BIT DATA attribute, ROWID, or a distinct type that is based on one of these types. The encoding applies to element content or attribute values.

XMLBINARY USING BASE64
Specifies that the assumed encoding is base64 characters, as defined for XML schema type xs:base64Binary. The base64 encoding uses a 65-character subset of US-ASCII (10 digits, 26 lowercase characters, 26 uppercase characters, '+' and '/') to represent every 6 bits of the binary or bit data by one printable character in the subset. These characters are selected so that they are universally representable. Using this method, the size of the encoded data is 33 percent larger than the original binary or bit data.

XMLBINARY USING BASE64 is the default.

XMLBINARY USING HEX
Specifies that the assumed encoding is hexadecimal characters as defined for XML schema type xs:hexBinary encoding. The hex encoding represents each byte (8 bits) with two hexadecimal characters. Using this method, the encoded data is twice the size of the original binary or bit data.

This function takes an element name, an optional collection of namespace declarations, an optional collection of attributes, and zero or more optional arguments that make up the content of the XML element. The result is an XML sequence that contains an XML element node or the null value. If the results of all *element-content-expression* arguments are empty strings, the result is an XML sequence that contains an empty element.

The result of the function is an XML value. The result can be null; if all *element-content-expression* arguments are null and the NULL ON NULL option is in effect, the result is the null value.

Constructing an element node: The resulting element node is constructed as follows:

1. *xmlnamespace-declaration* adds a set of in-scope namespaces for the constructed element. Each in-scope namespace associates a namespace prefix (or the default namespace) with a namespace URI. The in-scope namespaces define the set of namespace prefixes that are available for interpreting QNames within the scope of the element.
2. If the *xmlattributes-function* is specified, it is evaluated and the result is a sequence of attribute nodes.
3. Each *element-content-expression* is evaluated and the result is converted into a sequence of nodes as follows:
   - If the result type is not XML, it is converted to an XML text node that contains the result of the *element-content-expression* this is mapped to XML.
   - If the result type is XML, the result is a sequence of items. Some of the items in that sequence might be document nodes. Each document node in the sequence is replaced by the sequence of its top-level children. Then for each node in the resulting sequence, a new deep copy of the node is constructed, including its children and attributes. Each copied node has a new node identity. Copied element nodes are given the type annotation xdt:untyped, and copied attribute nodes are given the type annotation xdt:untypedAtomic. For each adjacent sequence of one or more atomic values that are returned in the sequence, a new text node is constructed that contains the result of casting each atomic value to a string, with a single blank character inserted between adjacent values. If any of these atomic values cannot be cast into a string, an error is returned.
4. The result sequence of *xmlattributes-function* and the resulting sequences of all *element-content-expression* clauses are concatenated into one sequence which is called the *content sequence*. Any sequence of adjacent text nodes in the content sequence is merged into a single text node by concatenating their contents, with no intervening blanks. After concatenation, any text node that is a zero-length string is deleted from the content sequence.
5. If the content sequence contains an attribute node that follows a node that is not an attribute node, an error is returned. Attribute nodes that occur in the content sequence become attributes of the new element node. If two or more of these attribute nodes have the same name, an error is returned. A namespace declaration is created that corresponds to any namespace that is used in the names of the attribute nodes if the namespace URI is not in the in-scope namespaces of the constructed element.
6. Element, text, comment, and processing instruction nodes in the content sequence become the children of the constructed element node.

7. The constructed element node is given a type annotation of xdt:untyped, and each of its attributes is given a type annotation of xdt:untypedAtomic. The node name of the constructed element node is the XML element name that is specified after the NAME keyword.

Rules for using namespaces within XMLELEMENT: The following rules describe scoping of namespaces:

- The namespaces that are declared in the XMLNAMESPACES function are the in-scope namespaces of the element node that are constructed by the XMLELEMENT function. If the element node is serialized, each of its in-scope namespaces will be serialized as a namespace attribute unless it is an in-scope namespace of the parent of the element node and the parent element is also serialized.
- The scope of these namespaces is the lexical scope of the XMLELEMENT function, including the element name, the attribute names that are specified in the XMLATTRIBUTES function, and all *element-content-expressions*. These are used to resolve the QNames in the scope.
- If an XMLQUERY or XMLEXISTS function is in an *element-content-expression*, the namespaces become the *statically known namespaces* of the XQuery expression of the XMLQUERY or XMLEXISTS function. Statically known namespaces are used to resolve the QNames that are in the XQuery expression. If the XQuery prolog declares a namespace that has the same prefix within the scope of the XQuery expression, the namespace that is declared in the prolog will override the namespaces that are declared in the XMLNAMESPACES function.
- If an attribute of the constructed element comes from *element-content-expression*, its namespace might not already be declared as an in-scope namespace of the constructed element. In this case, a new namespace is created for it. If the prefix of the attribute name is already bound to a different URI by a in-scope namespace, DB2 generates a different prefix to be used in the attribute name. A namespace is created for this generated prefix. The name of the generated prefix follows the following pattern: db2ns-*xx*, where *xx* is a pair of characters chosen from the set [A-Z,a-z,0-9].

Example 1: The following statement uses the XMLELEMENT function to create an XML element that contains an employees name. The statement also stores the employee number as an attribute named serial. If there is a null value in the referenced column, the function returns the null value:

```
SELECT e.empno, e.firstnme, e.lastname,
       XMLELEMENT ( NAME "foo:Emp",
         XMLNAMESPACES('http://www.foo.com' AS "foo"),
         XMLATTRIBUTES(e.empno as "serial"),
                       e.firstnme,
                       e.lastname
                       OPTION NULL ON NULL ) AS "Result"
    FROM EMP e
    WHERE e.edlevel = 12;
```

The result of the query would look similar to the following result:

```
 EMPNO  FIRSTNME  LASTNAME  Result
-----  --------  --------  -----------------------------------

A0001  John      Parker    <foo:Emp xmlns:foo="http://www.foo.com"
                                serial="A0001">JohnParker</foo:Emp>
B0001  (null)    Smith     <foo:Emp xmlns:foo="http://www.foo.com"
                                serial="B0001">Smith</foo:Emp>
B0002  (null)    (null)    (null)
(null) (null)    (null)    (null)
```

Example 2: The following example is similar to Example 1, however, when a null value is in one of the referenced columns, an empty element is returned:

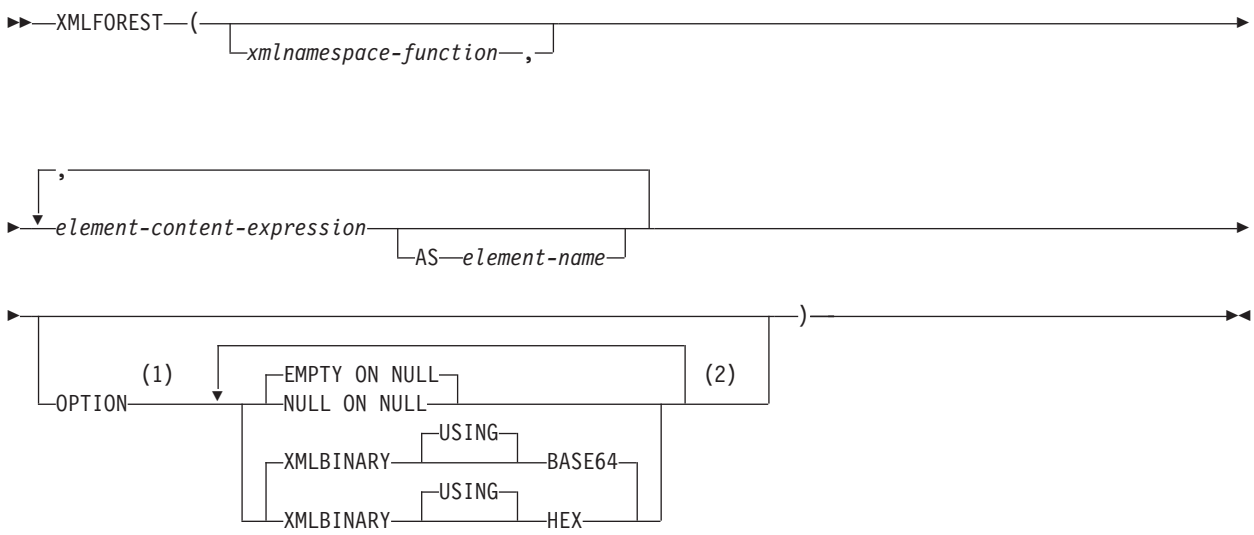
```
SELECT e.empno, e.firstnme, e.lastname,
  XMLELEMENT (NAME "foo:Emp",
              XMLNAMESPACES('http://www.foo.com' AS "foo"),
              XMLATTRIBUTES(e.empno as "serial"),
                            e.firstnme,
                            e.lastname
                         OPTION EMPTY ON NULL) AS "Result"
 FROM EMP e
 WHERE e.edlevel = 12;
```

The result of the query would look similar to the following result:

```
EMPNO    FIRSTNME    LASTNAME    Result
-----    --------    --------    -------------------------------------

A0001    John        Parker      <foo:Emp xmlns:foo="http://www.foo.com"
                                  serial="A0001">JohnParker</foo:Emp>
B0001    (null)      Smith       <foo:Emp xmlns:foo="http://www.foo.com"
                                  serial="B0001">Smith</foo:Emp>
B0002    (null)      (null)      <foo:Emp xmlns:foo="http://www.foo.com"
                                  serial="B0002"/>
(null)   (null)      (null)      <foo:Emp xmlns:foo="http://www.foo.com"/>
```

XMLFOREST

The XMLFOREST function returns an XML value that is a sequence of XML element nodes.



Notes:

1 The OPTION clause can only be specified if at least one *xmlattributes-function* or *element-content-expression* is specified.

2 The same clause must not be specified more than one time.

The schema is SYSIBM.

xmlnamespace-function
Specifies the XML namespace declarations that are the result of the XMLNAMESPACES function. The namespaces that are declared are in the scope of the XMLFOREST function. The namespaces apply to any nested XML functions within the XMLFOREST function, regardless of whether or not those functions appear inside another subselect. See "XMLNAMESPACES" on page 718 for more information on declaring XML namespaces.

If *xmlnamespace-function* is not specified, namespace declarations are not associated with the constructed sequence of XML element nodes.

element-content-expression
Specifies an expression that returns a value that is used for the content of a generated XML element. The result of the expression is mapped to an XML value according to the mapping rules from an SQL value to an XML value. If the expression is not a simple column reference, *element-name* must be specified.

AS *element-name*
Specifies an identifier that is used for the XML element name.

An XML element name must be an XML QName. If the name is qualified, the namespace prefix must be declared within the scope.

If *element-name* is not specified, *element-content-expression* must be a column name. The element name is created from the column name using the fully escaped mapping from a column name to a QName.

OPTION
Specifies options for the result for NULL values, binary data, and bit data. The options will not be inherited by the XMLELEMENT or XMLFOREST functions that appear in *element-content-expression*.

EMPTY ON NULL or NULL ON NULL
Specifies if a null value or an empty element is returned when the values of each *element-content-expression* is a null value. EMPTY ON NULL and NULL on NULL only affect null handling of the *element-content-expression* arguments, not the handling of values from an *xmlattributes-function* argument.

EMPTY ON NULL
If the value of each *element-content-expression* is null, an empty element is returned.

EMPTY ON NULL is the default.

NULL ON NULL
If the value of each *element-content-expression* is null, a null value is returned.

XMLBINARY USING BASE64 or XMLBINARY USING HEX
Specifies the assumed encoding of binary input data, character string data with the FOR BIT DATA attribute, ROWID, or a distinct type that is based on one of these types. The encoding applies to element content or attribute values.

XMLBINARY USING BASE64
Specifies that the assumed encoding is base64 characters, as defined for XML schema type xs:base64Binary encoding. The base64 encoding uses a 65-character subset of US-ASCII (10 digits, 26 lowercase characters, 26 uppercase characters, '+' and '/') to represent every 6 bits of the binary or bit data by one printable character in the subset. These characters are selected so that they are universally representable. Using this method, the size of the encoded data is 33 percent larger than the original binary or bit data.

XMLBINARY USING BASE64 is the default.

XMLBINARY USING HEX
Specifies that the assumed encoding is hexadecimal characters, as defined for XML schema type xs:hexBinary encoding. The hex encoding represents each byte (8 bits) with two hexadecimal characters. Using this method, the encoded data is twice the size of the original binary or bit data.

The XMLFOREST function can be expressed using the XMLCONCAT and XMLELEMENT functions.

This function takes an optional set of namespace declarations and one or more arguments that make up the name and element content for one or more element nodes. The result is an XML sequence containing a sequence of element nodes or the null value.

The result of the function is an XML value. The result can be null; if all the *element-content-expression* arguments are null and the NULL ON NULL option is in effect, the result is the null value.

Example: Generate an "Emp" element for each employee. Use employee name as its attribute and two subelements generated from columns HIRE and DEPT by using XMLFOREST as its content. The element names for the two subelements are "HIRE" and "department".

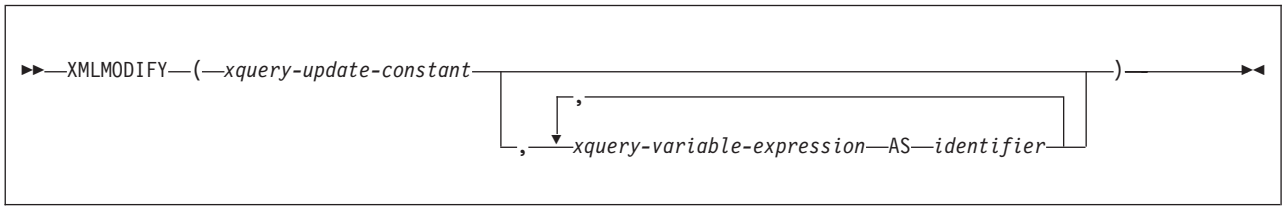
```
SELECT e.id, XMLSERIALIZE ( XMLELEMENT
       ( NAME "Emp",
         XMLATTRIBUTES ( e.fname || ' '  || e.lname
                         AS "name" ),
         XMLFOREST ( e.hire,
                     e.dept AS "department" )
                  ) ) AS "result"
FROM employees e;
```

The result of the query would be similar to the following result:

```
ID     result
---------------------------------------------

1001   <Emp name="John Smith">
         <HIRE>2000-05-24</HIRE>
         <department>Accounting</department>
           </Emp>
1001   <Emp name="Mary Martin">
         <HIRE>1996-02-01</HIRE>
         <department>Shipping</department>
           </Emp>
```

XMLMODIFY

The XMLMODIFY function returns an XML value that might have been modified by the evaluation of an XQuery updating expression and XQuery variables that are specified as input arguments.



The schema is SYSIBM.

xquery-update-constant
Specifies an SQL character string constant that is interpreted as an XQuery updating expression that uses supported XQuery language syntax. *xquery-update-constant* must be an insert expression, a delete expression, or a replace expression. *xquery-update-constant* must not be an empty string or a string of all blanks.

xquery-variable expression
xquery-variable-expression specifies an SQL expression whose value is available to the XQuery expression that is specified by *xquery-update-constant* during execution.

The data type of *xquery-variable-expression* can be XML, integer, decimal, or a character or graphic string that is not a LOB. *xquery-variable-expression* must not return a ROWID, TIMESTAMP, binary string, REAL, DECFLOAT data types, or a character string that is bit data, and *xquery-variable-expression* must not reference a sequence expression. If the result value is of type XML, it is passed by reference, which means that the original values, not copies, are used in the evaluation of the XQuery expression. A null XML value is converted to an XML empty sequence. If the resulting value is not of type XML, the result of the expression must be castable to an XML value. A null value is converted to an XML empty sequence. The non-XML values creates a new copy of the value during the cast to XML.

An XQuery variable is created for each *xquery-variable-expression* this is specified, and the XQuery variable is set to a value that is equal to the *input-xml-value*.

AS *identifier*
Specifies that the value that is generated by *xquery-variable-expression* is passed to *xquery-update-constant* as an XQuery variable named *identifier*. The length of the name must not be longer than 128 bytes. If the length of the name is longer than 128 bytes, an error is returned. The leading dollar sign ($) that precedes variable names in the XQuery language is not included in *identifier*. The name must be an XML 1.0 NCName that is not the same as the identifier for another *xquery-variable-expression* in the same PASSING clause. If the identifier is not an XML 1.0 NCName an error is returned. If more than one *xquery-variable expression* have the same name, an error is returned. If the result of an *xquery-variable expression* is null, an empty sequence is assigned to the corresponding XQuery variable.

The XMLMODIFY function can only be used in an SQL UPDATE statement or within the update clause of an SQL MERGE statement. The XMLMODIFY function must be the topmost expression on the right hand side of the SET assignment clause of the update.

The *target-xml-column* is the XML column in the SET assignment clause that is to be updated by the value that is returned by the XMLMODIFY function. The initial context item in the XQuery updating expression is the value of the *target-xml-column* that is passed by reference. Only the value of the *target-xml-column* can be modified by the XQuery updating expression. In other words, the target expression nodes in the XQuery updating expression must be a node in the value of *target-xml-column*. The *target-xml-column* must be an XML column that is defined in the XML versioning format.

The value of *target-xml-column* that is modified by the XQuery updating expression is returned by the function. If the value of *target-xml-column* is null, the function returns null. Otherwise, the result of the XMLMODIFY function must be a well-formed XML document. If the XQuery updating expressions makes no modifications to the value of *target-xml-column*, the unmodified XML value is returned by the function. The XMLMODIFY function preserves the original node identities and the document order of *target-xml-column*. Although XMLMODIFY modifies *target-xml-column* by reference, for each row that is updated by the SQL UPDATE statement, any reference to *target-xml-column* in the SQL UPDATE statement is the value of the *target-xml-column* before the row is updated.

Example 1: The following is an example of an XMLMODIFY function with an XQuery insert expression. Assume that a table contains a column named PO that contains an XML document, 'purchaseOrders':

```
UPDATE purchaseOrders
   SET PO = XMLMODIFY('declare namespace ipo="http://www.example.com/IPO";
                       declare namespace pyd="http://www.examplepayment.com";
                       insert node $payment/@pyd:paidDate
                         as first into /ipo:purchaseOrder/billTo',
                       XMLPARSE(DOCUMENT
            '<payment xmlns:pyd="http://www.examplepayment.com"
                                  pyd:paidDate="2000-01-07">278.94
             </payment>') AS "payment")
```

The result of the purchaseOrders XML document in the PO column will be as follows:

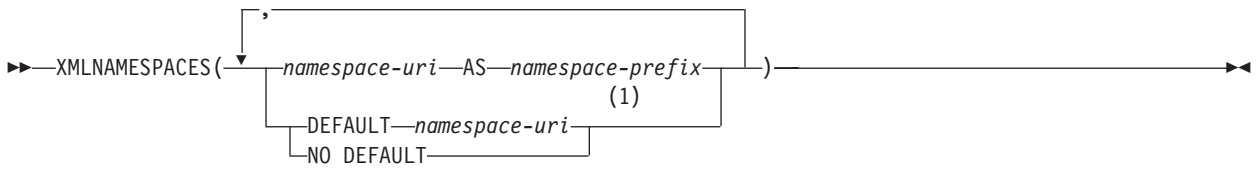
```
<ipo:purchaseOrder
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:ipo="http://www.example.com/IPO"
  xmlns:pyd="http://www.examplepayment.com"
  orderDate="1999-12-01" pyd:paidDate="2000-01-07">

<shipTo exportCode="1" xsi:type="ipo:UKAddress">
<name>Helen Zoe</name>
      <street>47 Eden Street</street>
      <city>Cambridge</city>
      <postcode>CB1 1JR</postcode>
  </shipTo>
  .
  .
  .
 </ipo:purchaseOrder>
```

Related concepts:

Basic updating expressions (DB2 Programming for XML)

XMLNAMESPACES

The XMLNAMESPACES function constructs namespace declarations from the arguments. This function can be used as an argument only for specific functions, such as the XMLELEMENT function and the XMLFOREST function.



Notes:

1 The DEFAULT or NO DEFAULT clause can only be specified one time.

The schema is SYSIBM.

The result is one or more XML namespace declarations containing in-scope namespaces for each non-null input value.

namespace-uri
Specifies an SQL character string constant that contains the namespace name or a universal resource identifier (URI). The character string constant must not be an empty string if it is used with *namespace-prefix*. *namespace-uri* cannot be `http://www.w3.org/XML/1998/namespace` or `http://www.w3.org/2000/xmlns/`.

AS *namespace-prefix*
Specifies a namespace prefix. The prefix is an SQL identifier that must be in the form of an XML NCName. The prefix must not be "xml" or "xmlns". The prefix must be unique within the list of namespace declarations.

The following namespace prefixes are pre-defined in SQL/XML: "xml", "xs", "xsd", "xsi", and "sqlxml". Their bindings are:

- xmlns:xml = "http://www.w3.org/XML/1998/namespace"
- xmlns:xs = "http://www.w3.org/2001/XMLSchema"
- xmlns:xsd = "http://www.w3.org/2001/XMLSchema"
- xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
- xmlns:sqlxml= "http://standards.iso.org/iso/9075/2003/sqlxml"

DEFAULT *namespace-uri* **or NO DEFAULT**
Specifies whether a default namespace is to be used within the scope of this namespace declaration.

The scope of this namespace declaration is the specified XML element and all XML expressions that are contained in the specified XML element.

DEFAULT *namespace-uri*
Specifies the default namespace to use within the scope of this namespace declaration. The *namespace-uri* applies for unqualified names in the scope unless it is overridden in a nested scope by another DEFAULT declaration or by a NO DEFAULT declaration.

namespace-uri specifies an SQL character string constant that contains a namespace name or universal resource identifier (URI). The character string constant can be an empty string in the context of the DEFAULT clause.

NO DEFAULT
Specifies that no default namespace is to be used within the scope of this namespace declaration. There is no default namespace in the scope unless the NO DEFAULT clause is overridden in a nested scope by a DEFAULT declaration.

The result of the function is an XML value that is an XML sequence that contains an XML namespace declaration for each specified namespace. The result cannot be null.

Example 1: Generate an "employee" element for each employee. The employee element is associated with XML namespace "urn:bo', which is bound to prefix "bo". The element contains attributes for names and a hiredate subelement.

```
SELECT e.empno, XMLSERIALIZE(XMLELEMENT(NAME "bo:employee",
                   XMLNAMESPACES('urn:bo' as "bo"),
                   XMLATTRIBUTES(e.lastname, e.firstnme),
                   XMLELEMENT(NAME "bo:hiredate", e.hiredate)) AS CLOB(50))
   FROM employee e where e.edlevel = 12;
```

The result of the query would be similar to the following result:

```
00029 <bo:employee xmlns:bo="urn:bo" LASTNAME="PARKER" FIRSTNME="JOHN">
      <bo:hiredate>198-5-3</bo:hiredate>
      </bo:employee>
00031 <bo:employee xmlns:bo="urn:bo" LASTNAME="SETRIGHT"
      FIRSTNME="MAUDE">
      <bo:hiredate>1964-9-12</bo:hiredate>
      </bo:employee>
```

Example 2: Generate two elements for each employee using XMLFOREST. The first "lastname" element is associated with the default namespace "http://hr.org", and the second "job" element is associated with XML namespace "http://fed.gov", which is bound to prefix "d".

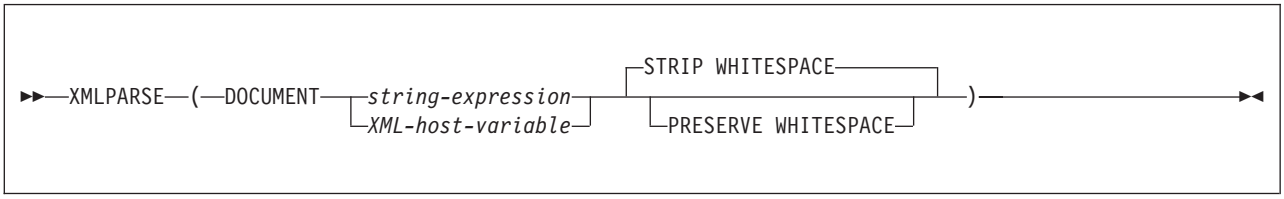
```
SELECT empno, XMLSERIALIZE(XMLFOREST(
          XMLNAMESPACES(DEFAULT 'http://hr.org', 'http://fed.gov' AS "d"),
         lastname, job AS "d:job") AS CLOB(50))
FROM employee where edlevel = 12;
```

The result of the query would be similar to the following result:

```
00029 <LASTNAME xmlns="http://hr.org" xmlns:d="http://fed.gov">PARKER
      </LASTNAME>
      <d:job xmlns="http://hr.org" xmlns:d="http://fed.gov">
      OPERATOR</d:job>
00031 <LASTNAME xmlns="http://hr.org" xmlns:d="http://fed.gov">
      SETRIGHT</LASTNAME>
      <d:job xmlns="http://hr.org" xmlns:d="http://fed.gov">
      OPERATOR</d:job>
```

XMLPARSE

The XMLPARSE function parses the argument as an XML document and returns an XML value.



The schema is SYSIBM.

DOCUMENT
Specifies that the character string expression to be parsed must evaluate to a well-formed XML document that conforms to XML 1.0.

string-expression
An expression that returns a character, graphic, or binary string.

string-expression must evaluate to a character string that conforms to the definition of a well-formed XML document as defined in XML 1.0.

XML-host-variable
An XML host variable that contains a well-formed XML document as defined in XML 1.0. *XML-host-variable* must not be binary XML data.

STRIP WHITESPACE or PRESERVE WHITESPACE
Specifies whether whitespace is to be removed or preserved. Any DTD attributes for xml:space have no impact on whitespace handling.

STRIP WHITESPACE
Specifies that whitespace (space that is between element nodes without any non-whitespace text nodes) will be stripped unless the nearest containing element has a value of 'preserve' for the xml:space attribute.

STRIP WHITESPACE is the default.

PRESERVE WHITESPACE
Specifies that all whitespace is preserved, even when the nearest containing element has a value of 'default' for the xml:space attribute.

The result of the function is XML. If *string-expression* can be null, the result can be null; if *string-expression* is null, the result is the null value.

Direct use of XMLPARSE with character string input: Applications should avoid direct use of the XMLPARSE function with character string input and should send strings that contain XML documents directly by using host variables to maintain the match between the external encoding and the encoding in the XML declaration. If XMLPARSE must be used in this situation, a BLOB type should be specified as the argument to avoid code page conversion.

Example 1: The following example inserts an XML document into the EMP table and preserves the whitespace in the original XML document. Assume that *hv* contains the value, '<a xml:space='preserve'> <c>c</c>b ':

```
INSERT INTO EMP (id, xvalue) VALUES(1001,
                  XMLPARSE(DOCUMENT :hv
                           PRESERVE WHITESPACE));
```

XMLPARSE will treat the value in *hv* for the insert statement as equivalent to the following value:

```
<a xml:space='preserve'> <b> <c>c</c>b </b>
</a>
```

Example 2: The following example inserts an XML document into the EMP table and strips the whitespace in the original XML document. Assume that *hv* contains the value, '<a xml:space='preserve'> <c>c</c>b ':

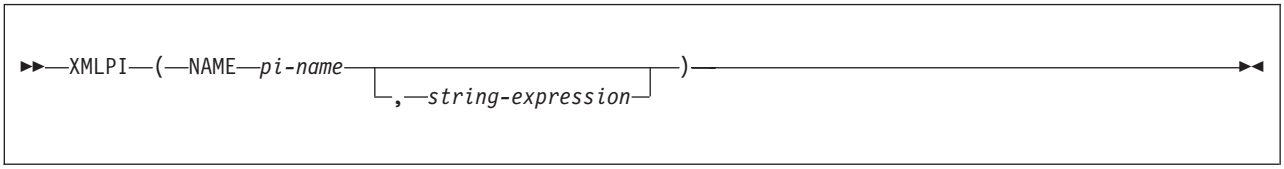
```
INSERT INTO EMP (id, xvalue) VALUES(1001,
                 XMLPARSE(DOCUMENT :hv
                          STRIP WHITESPACE));
```

XMLPARSE will treat the value in *hv* for the insert statement as equivalent to the following value:

```
<a xml:space='preserve'>
<b xml:space='default'><c>c</c>b </b>
</a>
```

XMLPI

The XMLPI function returns an XML value with a single processing instruction node.



The schema is SYSIBM.

NAME *pi-name*
Specifies the name of a processing instruction. The name is an SQL identifier that must be in the form of an XML NCName. The name must not contain "xml" in any case combination.

string-expression
An expression that returns a value of a built-in character or graphic string that is not a LOB and is not bit data The resulting string will be converted to UTF-8 and parsed to check for conformance to the content of XML processing instruction as specified by the following rules:

- The string must not contain the substring '?>' as this terminates a processing instruction.
- Each character can be any Unicode character, excluding the surrogate blocks, X'FFFE', and X'FFFF'.

If the resulting string does not conform to the preceding rules, an error is returned. The resulting string becomes the contents of the constructed processing instruction node. If *string-expression* is not specified or is an empty string, the contents of the procession instruction node are empty.

The result of the function is an XML value. The result can be null; if the *string-expression* argument is null, the result is the null value.

Example: Generate an XML processing instruction node:

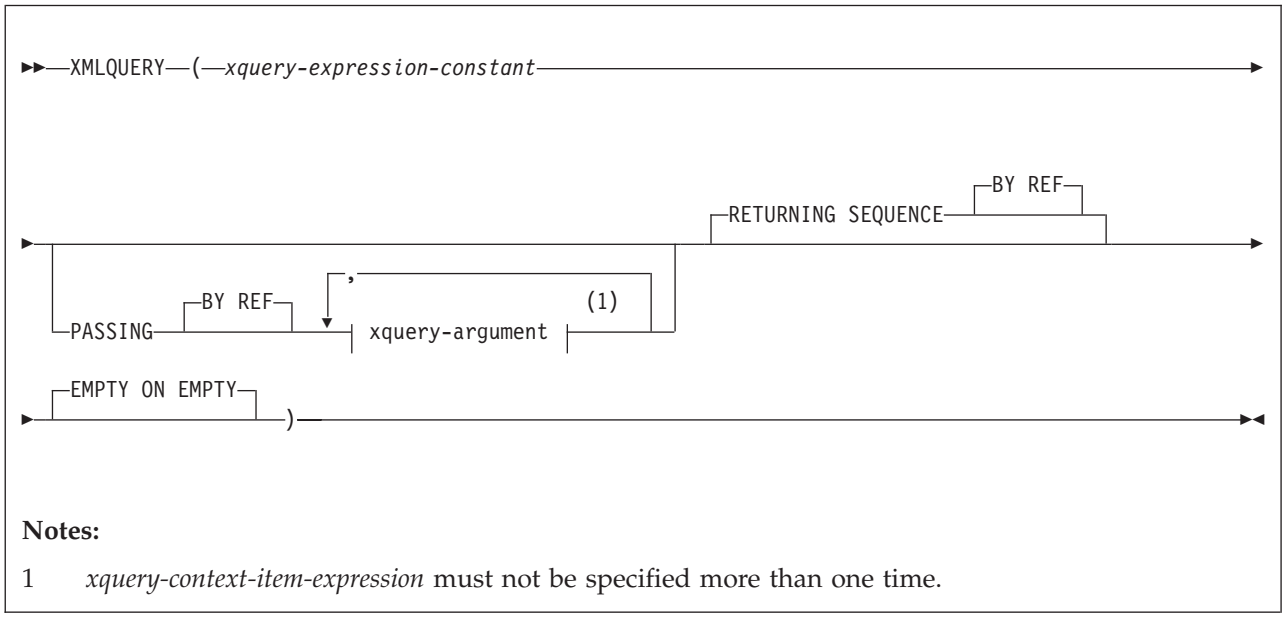
```
SELECT XMLPI(NAME "Instruction", 'Push the red button')
  FROM SYSIBM.SYSDUMMY1;
```

The result looks similar to the following results:

```
<?Instruction Push the red button?>
```

XMLQUERY

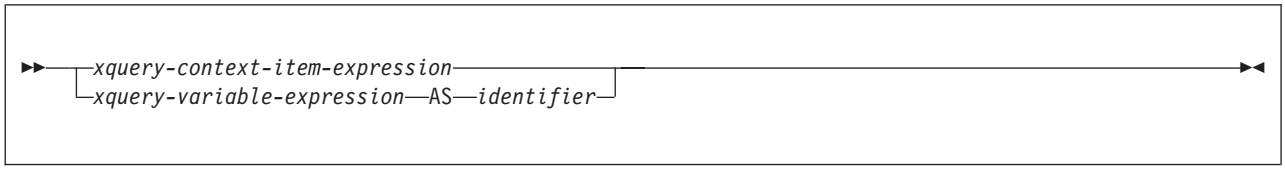
The XMLQUERY function returns an XML value from the evaluation of an XQuery expression, by using specified input arguments, a context item, and XQuery variables.



Notes:

1 *xquery-context-item-expression* must not be specified more than one time.

xquery-argument:



The schema is SYSIBM.

xquery-expression-constant
Specifies an SQL character string constant that is interpreted as an XQuery expression using supported XQuery language syntax. See *DB2 XML Guide* for information about the supported XQuery expressions. *xquery-expression-constant* cannot be an XQuery updating expression. The XQuery expression is evaluated with the arguments specified in *xquery-argument*, and returns an output sequence that is also returned as the result of the XMLQUERY function. *xquery-expression-constant* must not be an empty string or a string of all blanks.

PASSING
Specifies input values and the manner in which these values are passed to the XQuery expression that is specified by *xquery-expression-constant*.

BY REF
Specifies that the XML input value arguments are to be passed by reference. When XML values are passed by reference, the XQuery evaluation uses the input node trees which preserves all properties, including the original node identities and document order. If two arguments pass the same XML value, node identity comparisons and document ordering comparisons involving

some nodes that are contained between the two input arguments might refer to nodes that are within the same XML node tree.

BY REF has no impact on how non-XML values are passed. The non-XML values create a new copy of the value during the cast to XML.

xquery-argument
Specifies an argument that is passed to the XQuery expression that is specified by *xquery-expression-constant*. A query argument is an expression that returns a value that is XML, integer, decimal, or a character or graphic string that is not a LOB. *xquery-argument* must not return ROWID, TIMESTAMP, binary string, REAL, DECFLOAT data types, or a character string data type that is bit data, and must not reference a sequence expression.

xquery-argument specifies both a value and the manner in which that value is to be passed. How an argument in the PASSING clause is used in the XQuery expression depends on whether the argument is specified as *xquery-context-item-expression* or *xquery-variable-expression*. *xquery-argument* includes an SQL expression that is evaluated before passing the result to the XQuery expression.

- If the resulting value is of type XML, it becomes an *input-xml-value*. It is passed by reference, which means that the original values, not copies, are used in the evaluation of the XQuery expression. A null XML value is converted to an XML empty sequence.
- If the resulting value is not of type XML, the result of the expression must be able to be cast to an XML value. A null value is converted to an XML empty sequence. The converted value becomes an *input-xml-value*.

When *xquery-expression-constant* is evaluated, an XQuery variable receives a value that is equal to *input-xml-value* and a name as specified by the AS clause.

xquery-context-item-expression
xquery-context-item-expression specifies the initial context item in the XQuery expression specified by *xquery-expression-constant*. The value of the initial context item is the result of *xquery-context-item-expression* cast to XML. *xquery-context-item-expression* must not be specified more than one time.

xquery-context-item-expression must not be a sequence of more than one item. If *input-xml-value* is an empty XML string, the XQuery expression is evaluated with the initial context item set to an empty XML string. If the value of *input-xml-value* is null, the function returns a null value.

If the *xquery-context-item-expression* is not specified or is an empty sequence, the initial context item in the XQuery expression is undefined and the XQuery expression must not reference the initial context item.

An XQuery variable is not created for the context item expression.

xquery-variable-expression
xquery-variable-expression specifies an SQL expression whose value is available to the XQuery expression that is specified by *xquery-expression-constant* during execution. The sequence cannot contain a sequence reference.

An XQuery variable is created for each *xquery-variable-expression*, and the XQuery variable is set to a value equal to *input-xml-value*. For example, `PASSING T.A + T.B AS "sum"` creates an XQuery variable named sum. The scope of the XQuery variables that are created from the PASSING clause is the XQuery expression that is specified by *xquery-expression-constant*.

AS *identifier*
Specifies that the value that is generated by *xquery-variable-expression* is passed to *xquery-expression-constant* as an XQuery variable named *identifier*. The length of the name must not be longer than 128 bytes. The leading dollar sign ($) that precedes variable names in the XQuery language is not included in *identifier*. The name must be an XML 1.0 NCName that is not the same as the identifier for another *xquery-variable-expression* in the same PASSING clause.

RETURNING SEQUENCE
Specifies that the XQuery expression returns a sequence.

BY REF
Specifies that the result of the XQuery expression is returned by reference. If this value contains nodes, any expression that is using the return value of the XQuery expression will receive node references directly, preserving all node properties including the original node identities and document order.

EMPTY ON EMPTY
Specifies that an empty sequence that results from processing the XQuery expression is returned as an empty sequence.

The result of the function is an XML value. The result cannot be null.

If the evaluation of the XQuery expression results in an error, the XMLQUERY function returns the XQuery error.

Implicit casting of a non XML value to an XML value: If the result of *xquery-argument* is not an XML type, the value is cast to XML as follows. The SQL data type of the expression is mapped to a corresponding XML Schema data type according to the following table:

Table 80. SQL data types and corresponding XML schema data types

SQL data type	XML schema data type
CHAR, VARCHAR	xs:string
GRAPHIC, VARGRAPHIC	xs:string
SMALLINT	xs:integer
INTEGER	xs:integer
BIGINT	xs:integer
DECIMAL	xs:decimal
DOUBLE	xs:double
FLOAT	xs:double

Let *V* be the value of the expression. An atomic value of the corresponding XML schema data type is constructed such that the result of `cast (V as varchar)` is a lexical representation of the constructed atomic value. For example, an SQL VARCHAR value '123' is converted to an atomic value '123' of xs:string type. An SQL integer '12' is converted to an atomic value '12' of xs:integer. An SQL decimal value '1.20' is converted to an atomic value '1.2' of xs:decimal.

Example 1: The following example returns an XML value from evaluation of the specified XQuery expression:

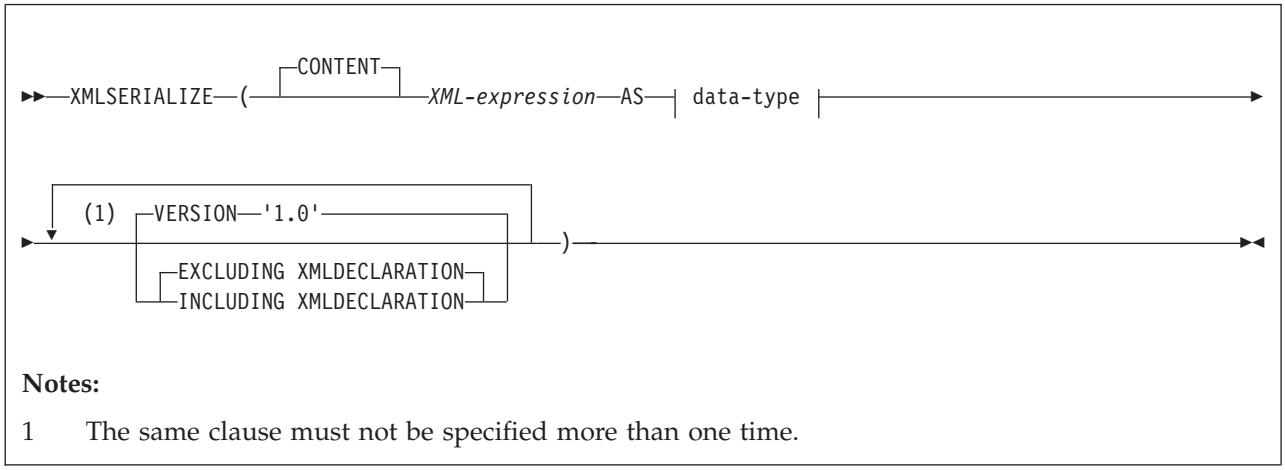
```
  SELECT XMLQUERY('//item[productName=$n]'
                  PASSING PO.POrder,
                  :hv AS "n") AS "Result"
     FROM PurchaseOrders PO;
```

Assume that the value of the host variable *hv* is 'Baby Monitor', the result is similar to the following results:

```
Result
--------------------------------------------------------------------------------

<item partNum="926-AA"><productName>Baby Monitor</productName><quantity>1
</quantity><USPrice>39.98</USPrice><shipDate>1999-05-21</shipDate></item>
```

XMLSERIALIZE

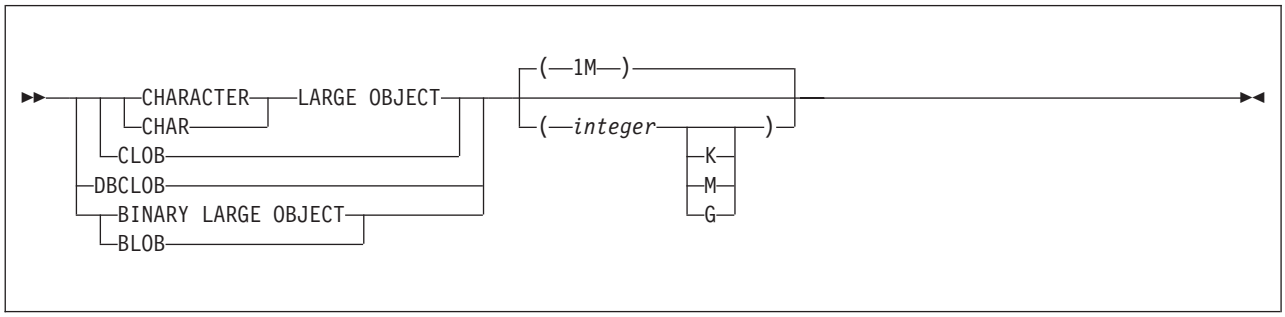
The XMLSERIALIZE function returns a serialized XML value of the specified data type that is generated from the first argument.



Notes:

1 The same clause must not be specified more than one time.

data-type



The schema is SYSIBM.

CONTENT
Specifies that any XML value can be specified and the result of the serialization is based on this input value.

XML-expression
An expression that returns an XML value that is not an attribute node. The atomic values in the input sequence must be able to be cast to xs:string. *XML-expression* is the input to the serialization process.

AS *data type*
Specifies the data type of the result. The implicit or explicit length attribute for the specified result data type must be sufficient to contain the serialized output.

The CCSID of a resulting character or graphic string is determined by the data type of the result:

- If the result is a CLOB, the CCSID for mixed Unicode data (1208).
- If the result is a DBCLOB, the CCSID for graphic Unicode data (1200).

VERSION '1.0'
Specifies the XML version of the serialized value. The only version that is supported is '1.0', which must be specified as a string constant.

EXCLUDING XMLDECLARATION or INCLUDING XMLDECLARATION
Specifies whether an XML declaration is included in the result.

EXCLUDING XMLDECLARATION
Specifies that an XML declaration is not included in the result.

EXCLUDING XMLDECLARATION is the default.

INCLUDING XMLDECLARATION
Specifies that an XML declaration is included in the result. The XML declaration contains values for XML serialization version 1.0 and an encoding specification of UTF-8. An XML sequence is effectively converted to have a single document node by applying the XMLDOCUMENT function to *XML-expression* prior to serializing the resulting XML nodes.

The data type and length attribute of the result are determined from the specified *data-type*. The result can be null; if the *XML-expression* argument is null, the result is the null value.

Serializing a sequence: The value of the input argument to XMLSERIALIZE is a sequence. Before a sequence is serialized, it is normalized. The purpose of sequence normalization is to create a sequence that can be serialized as a well-formed XML document or external general parsed entity, that also reflects the content of the input sequence to the extent possible. If the input sequence is an XML empty string, the result of serialization is an empty string. Otherwise, the result is constructed as follows:

- For each item in the sequence, if the item is atomic, the lexical representation of the item is obtained by casting it to an xs:string
- Each subsequence of adjacent strings in the sequence is merged into a single string with the values of the adjacent strings separated by a single space.
- For each item in the sequence, if the item is a string, a text node is created with a value that is equal to the string.
- For each node in the sequence, if the node is a document node, it is replaced it by its children.
- Each node must not be an attribute node.
- Each subsequence of adjacent text nodes in the sequence are merged into a single text node that with the values of the adjacent text nodes concatenated in order without a space between each node. Any text nodes of zero length are dropped.
- A document node is created and the sequence of nodes that was generated is copied as the children of the new document node.

Let *S* be any sequence, the normalization described in the preceding list is equivalent to `XMLDOCUMENT(S)`. Therefore, the following two expressions produce the same result:

- `XMLSERIALIZE(S AS CLOB)`
- `XMLSERIALIZE(XMLDOCUMENT(S) AS CLOB)`

Each instance of the following characters that appear in the content of a text node or in the value of an attribute node is mapped as following during serialization:

Character in content of text node	during serialization, the character is mapped to
'&' (X'26')	'&'
'<'(X'3C')	'<'
'>'(X'3E')	'>'
carriage return (X'0D')	'
'
quote (X'22')[1]	'"'

Note: The quote character is only mapped if it is inside of an attribute value.

Syntax alternatives: XML2CLOB(*XML-expression*) can be specified as an alternative to XMLSERIALIZE(*XML-expression* AS CLOB(2G)). XML2CLOB is supported only for compatibility with previous releases of DB2.

Example 1: Serialize into CLOB of UTF-8, the XML value that is returned by the XMLELEMENT function, which is a simple XML element with "Emp" as the element name, and an employee name as the element content:

```
SELECT e.id, XMLSERIALIZE(XMLELEMENT ( NAME "Emp",
                    e.fname || ' ' || e.lname)
             AS CLOB(100)) AS "result"
   FROM employees e;
```

The result looks similar to the following results:

```
ID    result
---   ----------------------------

1001  <Emp>John Smith</Emp>
1206  <Emp>Mary Martin</Emp>
```

Example 2: Serialize into a string of BLOB type, the XML value that is returned by the XMLELEMENT function:

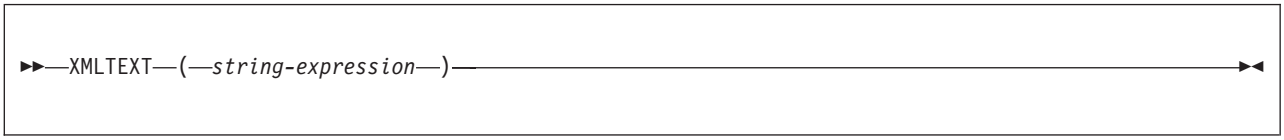
```
SELECT XMLSERIALIZE(XMLELEMENT(NAME "emp",
                      e.fname || ' ' || e.lname))
               AS BLOB(1K)
               VERSION '1.0') AS result
 FROM employee e WHERE e.id = '1001';
```

The result looks similar to the following results:

```
  result
------------------------

<emp>John Smith</emp>
```

XMLTEXT

The XMLTEXT function returns an XML value with a single text node that contains the value of the argument.



The schema is SYSIBM.

string-expression
An expression that returns a value of a built-in character or graphic string that is not bit data. Any character in the resulting string must be a valid XML 1.0 character when it is converted to UTF-8.

If *string-expression* is an empty string, an empty text node is returned.

The result of the function is an XML value.

The result can be null; if the argument is null, the result is the null value.

Example 1: The following example returns an XML value with a single text node that contains the specified value:

```
SELECT XMLTEXT('The stock symbol for Johnson&Johnson is JNJ.') AS "Result"
   FROM SYSIBM.SYSDUMMY1;
```

The result looks similar to the following results:

```
Result
--------------------------------------------------

The stock symbol for Johnson&amp;Johnson is JNJ.
```

Example 2: The XMLTEXT function enables the XMLAGG function to construct mixed content, as in the following example:

```
SELECT XMLELEMENT(NAME "para",
        XMLAGG(XMLCONCAT( XMLTEXT( plaintext),
                          XMLELEMENT( NAME "emphasis",
                          emphtext ))
                  ORDER BY seqno ), '.' ) as "result"
   FROM T;
```

Suppose that the content of the table T is as the following example:

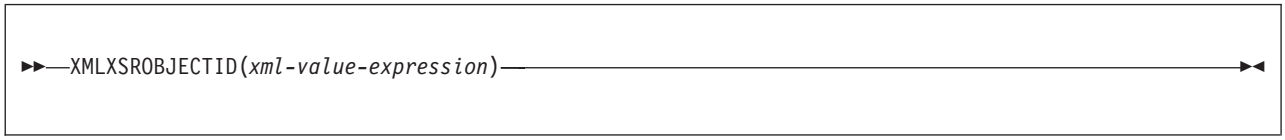
```
seqno    plaintext                                        emphtext
-----    -------------------------------------------------  ----------------
1        This query shows how to construct                mixed content
2        using XMLAGG and XMLTEXT. Without                XMLTEXT
3        XMLAGG cannot group text nodes with other nodes,   mixed content
         therefore, cannot generate
```

The result looks like the following result:

```
result
------------------------------------------------------------------------
<para>This query shows how to construct <emphasis>mixed content</emphasis>
using XMLAGG and XMLTEXT. Without <emphasis>XMLTEXT</emphasis>, XMLAGG
cannot group text nodes with other nodes, therefore, cannot generate
<emphasis>mixed content</emphasis>.</para>
```

XMLXSROBJECTID

The XMLXSROBJECTID function returns the XSR object identifier of the XML schema that is used to validate the XML document specified in the argument.



The schema is SYSIBM.

xml-value-expression
An expression that results in a value with a data type of XML. The resulting XML value must be an XML sequence with a single item that is an XML document or the null value.

The XSR object identifier is returned as a BIGINT value and provides the key to a single row in the SYSIBM.XSROBJECTS table.

The result can be null; if the argument is null, the result is the null value.

If *xml-value-expression* does not specify a validated XML document, the function returns 0.

Note: The XML schema that corresponds to an XSR object ID returned by the function might no longer exist, because an XML schema can be dropped without affecting XML values that were validated using that XML schema. Therefore, queries that use the XSR object ID to fetch further XML schema information from the SYSIBM.XSROBJECTS table might return an empty result set.

Example 1: Use the XMLXSROBJECTID function in conjunction with the DSN_XMLVALIDATE function to find all XML documents that are not validated in a table and validate them:

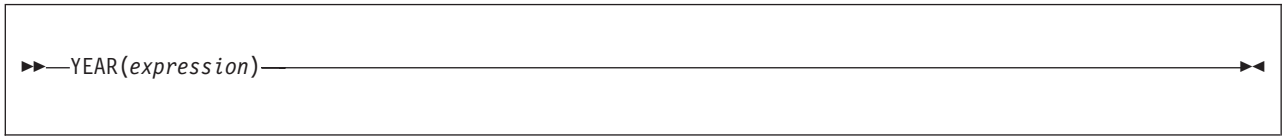
```
UPDATE orders
  SET content = dsn_xmlvalidate(content, 'SYSXSR.PO1')
 WHERE XMLXSROBJECTID(content) = 0;
```

Example 2: Use the XMLXSROBJECTID function to find the names and target namespaces of the XML schemas that are used to validate the XML documents in a table:

```
SELECT DISTINCT s.XSROBJECTNAME, s.targetNamespace
 FROM orders o, XSROBJECTS s
 WHERE XMLXSROBJECTID(content) = s.XSROBJECTID;
```

YEAR

The YEAR function returns the year part of a value that is a character or graphic string. The value must be a valid string representation of a date or timestamp.



The schema is SYSIBM.

The argument must be an expression that returns one of the following built-in data types: a date, a timestamp, a character string, a graphic string, or a numeric data type.

- If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp with an actual length of not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String representations of datetime values" on page 101.
- If *expression* is a number, it must be a date or timestamp duration. For the valid formats of date and timestamp durations, see "Datetime operands" on page 147.

If *expression* is a timestamp with a time zone, or a valid string representation of a timestamp with a time zone, the result is determined from the UTC representation of the datetime value.

The result of the function is a large integer.

The result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument specified:

If the argument is a date, a timestamp, or a string representation of either, the result is the year part of the value, which is an integer between 1 and 9999.

If the argument is a date duration or a timestamp duration, the result is the year part of the value, which is an integer between -9999 and 9999. A nonzero result has the same sign as the argument.

If the argument contains a time zone, the result is the year part of the value expressed in UTC.

Example 1: From the table DSN8B10.EMP, select all rows for employees who were born in 1941.

```
SELECT *
  FROM DSN8B10.EMP
  WHERE YEAR(BIRTHDATE) = 1941;
```

Example 2: The following invocations of the YEAR function returns the same result:

```
SELECT YEAR('1993-08-10-20.00.00'),
   YEAR('1993-08-10-20.00.00-08:00'),
   YEAR('1993-08-10-20.00.00+09:00')
  FROM SYSIBM.SYSDUMMY1;
```

For each invocation of the YEAR function in this SELECT statement, the result is 1993.

When the input argument contains a time zone, the result is determined from the UTC representation of the input value. The string representations of a timestamp with a time zone in the SELECT statement all have the same UTC representation: '1993-08-10-20.00.00'.

Table functions

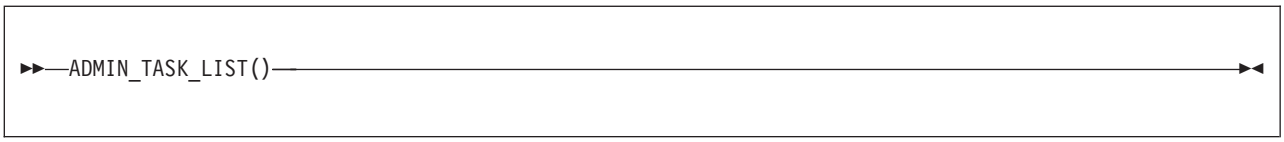
A table function can be used only in the FROM clause of a statement. Table functions return columns of a table and resemble a table created through a CREATE TABLE statement. Table functions can be qualified with a schema name.

ADMIN_TASK_LIST

The ADMIN_TASK_LIST function returns a table with one row for each of the tasks that are defined in the administrative task scheduler task list.

Authorization

The user who calls this function must have MONITOR1 privilege.



The schema is DSNADM.

The result of the function is a table with the format shown in the following table. All the columns are nullable except TASK_NAME.

Table 81. Format of the resulting table for ADMIN_TASK_LIST

Column name	Data type	Contains
BEGIN_ TIMESTAMP	TIMESTAMP	Contains the timestamp of when the task can first run. When the task begins to run depends on what values this and other columns contain: • If BEGIN_TIMESTAMP contains a non-null value: – If POINT_IN_TIME and TRIGGER_TASK_NAME contain null values, the task begins to run at the timestamp in BEGIN_TIMESTAMP – If POINT_IN_TIME contains a non-null value, the task begins to run at the next point in time that is defined at or after the timestamp in BEGIN_TIMESTAMP – If TRIGGER_TASK_NAME is a non-null value, the task begins to run at the next time that the task identified in TRIGGER_TASK_NAME completes or after the timestamp in BEGIN_ TIMESTAMP • If BEGIN_TIMESTAMP contains a null value: – If POINT_IN_TIME and TRIGGER_TASK_NAME contain null values, the task begins to run immediately – If POINT_IN_TIME contains a non-null value, the task begins to run at the next point in time that is defined – If TRIGGER_TASK_NAME is a non-null value, the task begins to run at the next time that the task identified in TRIGGER_TASK_NAME completes
END_ TIMESTAMP	TIMESTAMP	Contains the timestamp of when the task is last able to run. If this column is NULL, there are no restrictions as to when the task must not run.

Table 81. Format of the resulting table for ADMIN_TASK_LIST (continued)

Column name	Data type	Contains
MAX_ INVOCATIONS	INTEGER	Contains the maximum number of times the task can run. The maximum number applies to all types of schedules: triggered by events, scheduled by time interval, or by point in time. If this column is null, the task has no limit on the number of times it can be run. If both END_TIMESTAMP and MAX_INVOCATIONS contain values, the value in END_TIMESTAMP takes precedence over the value for MAX_INVOCATIONS. That is, if the value in END_TIMESTAMP is reached, even though the number of times the task has run has not reached the value for MAX_INVOCATIONS, the task will not run again
INTERVAL	INTEGER	Contains an integer that indicates the duration between the start of one instance of a task and the start of the next instance of the same task. If the value of this column is NULL, the task is not scheduled to run at a regular interval.
POINT_IN_ TIME	VARCHAR(400)	Contains one or more points in time (in UNIX cron format) for which the task is scheduled to run. If the value of this column is NULL, the task is not scheduled to run at a specific point in time. The format contains the following pieces of information separated by blanks: given hour, given minute, given day of the week, given day of the month, given month of the year.
TRIGGER_ TASK_NAME	VARCHAR(128)	Contains the task name of the task that, when its execution is complete, will trigger the running of the task that is described in the row. Task name DB2STOP is reserved for DB2 stop events and task name DB2START is reserved for DB2 start events. Those events are handled by the administrative task scheduler that is associated with the DB2 subsystem that is starting or stopping. If the value of this column is NULL, the task that is described in this row will not be triggered to run by another task.

Table 81. Format of the resulting table for ADMIN_TASK_LIST (continued)

Column name	Data type	Contains
TRIGGER_ TASK_COND	CHAR(2)	Contains the type of comparison that is to be made to the return code after the running of task that is indicated in TRIGGER_TASK_NAME. The following values are possible: **GT** Greater than **GE** Greater than or equal to **EQ** Equal to **LT** Less than **LE** Less tan or equal to **NE** Not equal to If this column contains NULL, the task is triggered to run without consideration of the return code of the task that is indicated in TRIGGER_TASK_NAME.
TRIGGER_ TASK_CODE	INTEGER	Contains the return code from running the task indicated in TRIGGER_TASK_NAME. If the running of this task is triggered by a stored procedure, TRIGGER_TASK_CODE contains the SQLCODE that must be returned by the stored procedure in order for this task to run. If the running of this task is triggered by a JCL job, TRIGGER_TASK_CODE contains the MAXRC that must be returned by the job in order for this task to run. "ADMIN_TASK_STATUS" on page 741 returns the SQLCODE or MAXRC value in the SQLCODE or MAXRC column. If TRIGGER_TASK_COND is NULL, this column will also be NULL.

Table 81. Format of the resulting table for ADMIN_TASK_LIST (continued)

Column name	Data type	Contains
DB2_SSID	VARCHAR(4)	Contains the DB2 subsystem ID of the DB2 subsystem that is associated with the administrative task scheduler that should run this task. The value in this column is used in a data sharing environment where, for example different DB2 members have different configurations and running the task relies on a certain environment. A value in DB2_SSID will prevent an administrative scheduler of other members to run this task, so that the task can only be run as long as the administrative task scheduler of the subsystem indicated in DB2_SSID is running. For a task that is being triggered by a DB2 start or DB2 stop event as indicated in the TRIGGER_TASK_NAME column, a value in DB2_SSID will allow the task to be run only when the indicated subsystem is starting or stopping. If no value is indicated in DB2_SSID, each subsystem that starts or stops will trigger a the task to be run locally, provided that the triggered task is run serially. If this column is NULL, any administrative scheduler can run this task.
PROCEDURE_ SCHEMA	VARCHAR(128)	Contains the schema of the DB2 stored procedure that this task will run. If the value of this column is null, DB2 uses a default schema.
PROCEDURE_ NAME	VARCHAR(128)	Contains the name of the DB2 stored procedure that this task will run. If the value of this column is NULL, no stored procedure will be called when this task is run.
PROCEDURE_ INPUT	VARCHAR(4096)	Contains a statement that returns one row of data. The returned value will be used as the input parameter of the stored procedure that this task will run. If this column contains the null value, no parameters are passed to the stored procedure when this task is run.
JCL_LIBRARY	VARCHAR(44)	Contains the name of the data set that contains the JCL job that is run when this task is run. If the value of this column is the null value, no JCL job will be run when this task is run.
JCL_MEMBER	VARCHAR(8)	Contains the name of the library member that contains the JCL job that is run when this task is run. If the value of this column is the null value, the data set that is specified in JCL_LIBRARY is sequential and contains the JCL job that is run when this task is run.

Table 81. Format of the resulting table for ADMIN_TASK_LIST (continued)

Column name	Data type	Contains
JOB_WAIT	VARCHAR(8)	Contains one of the following values, which indicates whether the JCL job can be run synchronously. If the value in the column is not null, this column contains one of the following values: **NO** Runs asynchronously **YES** Runs synchronously **PURGE** Runs synchronously and then the job status in z/OS is purged
TASK_NAME	VARCHAR(128)	Contains the unique name that is assigned to this task.
DESCRIPTION	VARCHAR(128)	Contains a description of the task if one exists.
USERID	VARCHAR(128)	Contains the authorization ID of the user under which the task will be invoked. If this column is NULL, the task is invoked by the default authorization ID that is associated with the administrative task scheduler.
CREATOR	VARCHAR(128)	Contains the authorization ID that added the task to the administrative task scheduler task list.
LAST_MODIFIED	TIMESTAMP	Timestamp of when the task was added or last modified.

Example 1: Retrieve information about all of the tasks that are defined in the administrative task scheduler task list:

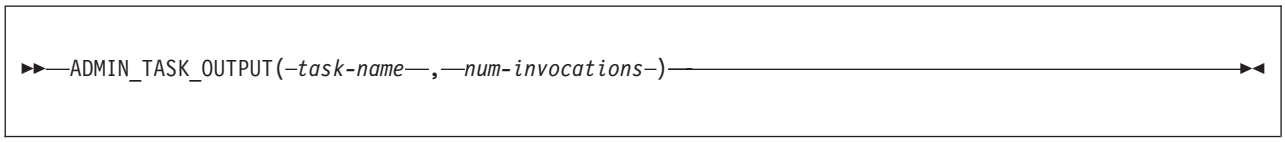
```
SELECT *
  FROM TABLE (DSNADM.ADMIN_TASK_LIST()) AS T;
```

ADMIN_TASK_OUTPUT

For an execution of a stored procedure, the ADMIN_TASK_OUTPUT function returns the output parameter values and result sets, if available. If the task that was executed is not a stored procedure or the requested execution status is not available, the function returns an empty table.

Authorization

The user who calls this function must have MONITOR1 privilege.



The schema is DSNADM.

Important: The ADMIN_TASK_OUTPUT function returns as many output parameter values and result sets as possible. However, this information is not always available. The administrative task scheduler cannot store output that exceeds 32,180 bytes in length. Therefore, some output parameters and result set values might be null if the values are too long to be stored by the administrative task scheduler. Also, if the result sets are too large to be stored, only some of the most recent rows of each result set might be available (for example, the first rows missing).

task-name
Specifies the unique name of the task whose execution output you want returned. This is an input parameter of type VARCHAR(128).

num-invocations
Specifies the execution number of the task whose output you want returned. This value must be a valid value in the NUM_INVOCATIONS column of the returned table of DSNADM.ADMIN_TASK_STATUS(NULL) for the specified task. This is an input parameter of type INTEGER.

The result of the function is a table with the format shown in the following table. This function might return an empty table for the output of a stored procedure for the following reasons:

- The stored procedure does not have output parameters or result sets.
- The output of the stored procedure was not stored at execution time, because the SYSIBM.ADMIN_TASKS_HIST table was not available.
- The *num-invocations* parameter is not valid.
- The output for the task that is specified by the *num-invocations* parameter is no longer stored, because the task is older than the value that is specified for the MAXHIST parameter of the administrative task scheduler. (The MAXHIST parameter specifies the maximum number of execution statuses to keep for each task.)

Table 82. Format of the resulting table for ADMIN_TASK_OUTPUT

Column name	Data type	Contains
RESULT_SET	SMALLINT	Contains the stored procedure result set number with a value beginning at 1, or NULL if this value is for an output parameter of the stored procedure.
ROW	SMALLINT	Contains the result set row number with a value beginning at 1, or NULL if this value is for an output parameter of the stored procedure.
COLUMN	SMALLINT	Contains the result set column number, or the index of an output parameter of the stored procedure parameters, with a value beginning at 1. Only the values of output parameters are returned, and the results include the index in all parameters of the stored procedure.
TYPE	CHAR(8)	Contains the type of the returned string. Possible types are: • DATE • TIME • TIMESTMP • CHAR • VARCHAR • FLOAT • BIGINT • INTEGER • SMALLINT • OTHER The value OTHER includes all of the other DB2 data types that are not supported in this stored procedure.
VALUE	VARCHAR(32180)	Contains the string representation of the output parameter value or the result set column value. This column is null if the TYPE column contains OTHER.

Related tasks:

Displaying the results of a stored procedure task (DB2 Administration Guide)

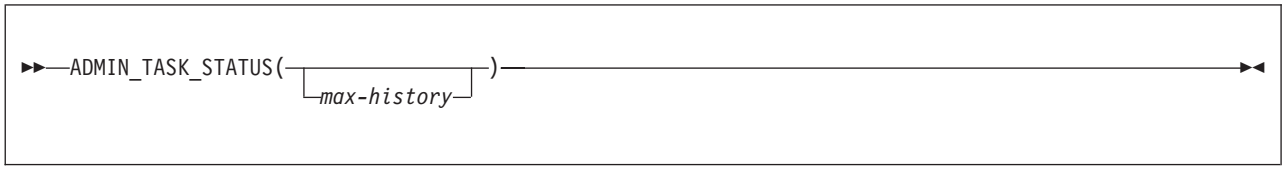
ADMIN_TASK_STATUS

The ADMIN_TASK_STATUS function returns a table with one row for each task that is defined in the administrative task scheduler task list. Each row indicates the status of the task for the last time it was run.

Optionally, if you specify the *max-history* parameter, the function returns a row of data for each execution of a task (up to the *max-history* value). For tasks that have not been executed, this function returns a row of data with a NULL status.

Authorization

The user who calls this function must have MONITOR1 privilege.



The schema is DSNADM.

max-history
Specifies the maximum number of execution statuses per task to be returned. The most recent execution statuses are returned.

If the parameter is set to NULL, all available task execution statuses are returned. If the parameter is set to 1, only the status for the last time the task was run is returned, which is the same result as not specifying this option. This is an input parameter of type INTEGER.

The result of the function is a table with the format shown in the following table.

Table 83. Format of the resulting table for ADMIN_TASK_STATUS

Column name	Data type	Contains
TASK_NAME	VARCHAR(128)	Contains the name of the task that has run, is running, or has been bypassed.

Table 83. Format of the resulting table for ADMIN_TASK_STATUS (continued)

Column name	Data type	Contains
STATUS	VARCHAR(10)	Contains one of the following values that indicates task status: **RUNNING** The task is currently running **COMPLETED** The task has finished running. For asynchronous tasks (JCL jobs), this column contains COMPLETED whenever the job is submitted to be run. Otherwise, this column contains COMPLETED only after the task has finished running. **NOTRUN** The task was not run at the scheduled invocation time. The MSG column contains the error or warning message that indicates why the task was not run. **UNKNOWN** The scheduler shut down while the task was running. The scheduler is started again but cannot know the execution status of this interrupted task.
NUM_ INVOCATIONS	INTEGER	Contains the number of times the administrative task scheduler attempted to run the task, including the current time if the task is currently running. The values in this column does not indicate if the task was successfully run.
START_ TIMESTAMP	TIMESTAMP	Contains the time when the task started running if the STATUS column contains COMPLETED, RUNNING, or UNKNOWN. Otherwise, this column contains the time that the task should have started to run but could not.
END_ TIMESTAMP	TIMESTAMP	Contains the time when the task finished running.
JOB_ID	CHAR(8)	Contains the job ID that is assigned to the JCL job submitted by the administrative task scheduler. This column contains NULL if the task is a stored procedure or if the STATUS column does not contain COMPLETED.
MAXRC	INTEGER	Contains the highest return code from submitting a JCL job. If the task is synchronous, the value in this column is changed to the return code that is returned when the job finishes running. This column is set to NULL if the task is a stored procedure, if the STATUS column does not contain COMPLETED, or if a synchronous task is finished and has run with JES3 in a z/OS 1.7 or earlier system.

Table 83. Format of the resulting table for ADMIN_TASK_STATUS (continued)

Column name	Data type	Contains
COMPLETION_ TYPE	INTEGER	Contains one of the following values that indicates the completion type of the JCL job submitted by the administrative task scheduler: **0** No completion information **1** Job ended normally **2** Job ended by completion code **3** Job had a JCL error **4** Job was canceled **5** Job abended **6** Converter abended while processing the job **7** Job failed security checks **8** Job failed in end-of-memory This column contains NULL if the task is a stored procedure, if the STATUS column does not contain COMPLETED, or if the JCL job is run with JES3 in a z/OS 1.7 or earlier system.
SYSTEM_ ABENDCD	INTEGER	Contains the system abend code returned by a failed JCL job that was submitted by the administrative task scheduler. This column contains NULL if the task is a stored procedure, if the STATUS column does not contain COMPLETED, or if the JCL job is run with JES3 in a z/OS 1.7 or earlier system.
USER_ABENDCD	INTEGER	Contains the user abend code returned by a failed JCL job that was submitted by the administrative task scheduler. This column contains NULL if the task is a stored procedure, if the STATUS column does not contain COMPLETED, or if the JCL job is run with JES3 in a z/OS 1.7 or earlier system.
MSG	VARCHAR(128)	Contains the error or warning message from the last time the task was run.
SQLCODE	INTEGER	Contains the SQLCODE set by DB2 when a stored procedure was called by the administrative task scheduler. This column contains NULL if the task is a JCL job or if the STATUS column does not contain COMPLETED.
SQLSTATE	CHAR(5)	Contains the SQLSTATE set by DB2 when a stored procedure was called by the administrative task scheduler. This column contains NULL if the task is a JCL job or if the STATUS column does not contain COMPLETED.

Table 83. Format of the resulting table for ADMIN_TASK_STATUS (continued)

Column name	Data type	Contains
SQLERRP	VARCHAR(8)	Contains the SQLERRP set by DB2 when a stored procedure was called by the administrative task scheduler. This column contains NULL if the task is a JCL job or if the STATUS column does not contain COMPLETED.
SQLERRMC	VARCHAR(70)	Contains the SQLERRMC set by DB2 when a stored procedure was called by the administrative task scheduler. This column contains NULL if the task is a JCL job or if the STATUS column does not contain COMPLETED.
DB2_SSID	VARCHAR(4)	Contains the DB2 subsystem ID that is associated with the administrative task scheduler that ran the task or should have run the task.
USERID	VARCHAR(128)	Contain the user ID that the task ran under.

Example 1: Retrieve status information about all of the tasks that have run in the administrative task scheduler task list:

```
SELECT *
  FROM TABLE (DSNADM.ADMIN_TASK_STATUS()) AS T;
```

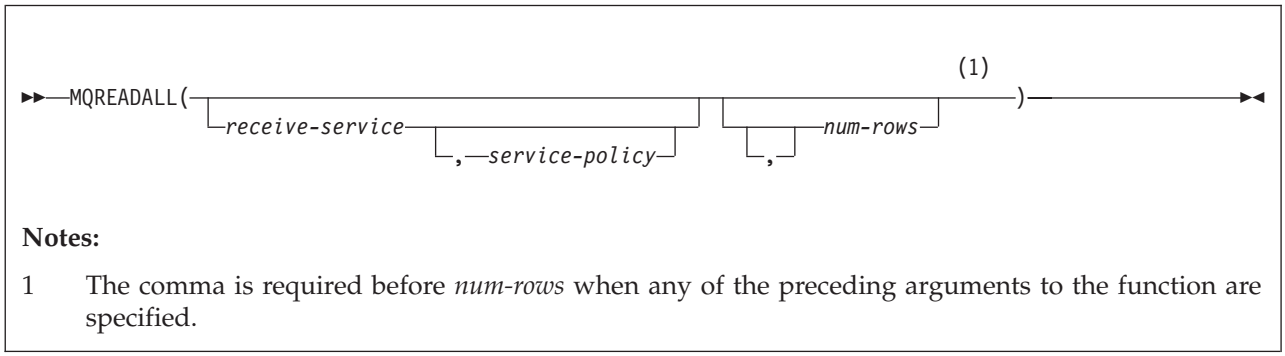
Related tasks:

Listing the last execution status of scheduled tasks (DB2 Administration Guide)

Listing multiple execution statuses of scheduled tasks (DB2 Administration Guide)

MQREADALL

The MQREADALL function returns a table that contains the messages and message metadata from a specified MQSeries location without removing the messages from the queue.



Notes:

1 The comma is required before *num-rows* when any of the preceding arguments to the function are specified.

The schema is DB2MQ.

The MQREADALL function returns a table containing the messages and message meta-data from the MQSeries location that is specified by *receive-service*, using the quality-of-service policy that is defined in *service-policy*. Performing this operation does not remove the messages from the queue that is associated with *receive-service*.

receive-service
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service point that is defined in the DB2MQ.MQSERVICE table. A service point is a logical end-point from which a message is sent or received. A service point definition includes the name of the MQSeries queue manager and the name of the queue. See *MQSeries Application Messaging Interface* for more details.

If *receive-service* is not specified or is the null value, DB2.DEFAULT.SERVICE is used.

service-policy
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service policy that is defined in the DB2MQ.MQPOLICY table. A service policy specifies a set of quality-of-service options that are to be applied to this messaging operation. These options include message priority and message persistence. See *MQSeries Application Messaging Interface* for more details.

If *service-policy* is not specified or is the null value, DB2.DEFAULT.POLICY is used.

num-rows
An expression that returns a value that is a SMALLINT or INTEGER data type whose value is a positive integer or zero. The value of the expression specifies the maximum number of messages to return.

If *num-rows* is not specified or if the value of the expression is zero, all available messages are returned.

The result of the function is a table with the format shown in the following table. All the columns are nullable.

Table 84. Format of the resulting table for MQREADALL

Column name	Data type	Contains
MSG	VARCHAR(4000)	The contents of the MQSeries message
CORRELID	VARCHAR(24)	The correlation ID that is used to relate messages
TOPIC	VARCHAR(40)	The topic that the message was published with, if available
QNAME	VARCHAR(48)	The name of the queue from which the message was received
MSGID	CHAR(24)	The unique, MQSeries-assigned identifier for the message
MSGFORMAT	VARCHAR(8)	The format of the message, as defined by MQSeries

Example 1: Read all the messages from the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY).

```
SELECT *
  FROM SYSIBM.SYSDUMMY1 (MQREADALL()) AS T;
```

The messages and all the metadata are returned as a table.

Example 2: Read all the messages from the beginning of the queue specified by the service MYSERVICE, using the default policy (DB2.DEFAULT.POLICY).

```
 SELECT T.MSG, T.CORRELID
FROM SYSIBM.SYSDUMMY1 (MQREADALL ('MYSERVICE')) AS T;
```

Only the MSG and CORRELID columns are returned.

Example 3: Read all the messages from the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY). Only messages with a CORRELID of '1234' are returned.

```
SELECT *
  FROM SYSIBM.SYSDUMMY1 (MQREADALL(10)) AS T
WHERE T.CORRELID = '1234';
```

All columns are returned.

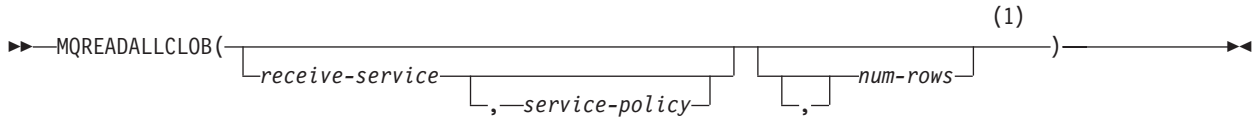
Example 4: Retrieve the first 10 messages from the beginning of the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY).

```
SELECT *
  FROM SYSIBM.SYSDUMMY1 (MQREADALL(10)) AS T;
```

The first 10 messages and all the columns are returned as a table.

MQREADALLCLOB

The MQREADALLCLOB function returns a table that contains the messages and message metadata from a specified MQSeries location without removing the messages from the queue.



Notes:

1 The comma is required before *num-rows* when any of the preceding arguments to the function are specified.

The schema is DB2MQ.

The MQREADALLCLOB function returns a table containing the messages and message meta-data from the MQSeries location that is specified by *receive-service*, using the quality-of-service policy that is defined in *service-policy*. Performing this operation does not remove the messages from the queue that is associated with *receive-service*.

receive-service
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service point that is defined in the DB2MQ.MQSERVICE table. A service point is a logical end-point from which a message is sent or received. A service point definition includes the name of the MQSeries queue manager and the name of the queue. See *MQSeries Application Messaging Interface* for more details.

If *receive-service* is not specified or is the null value, DB2.DEFAULT.SERVICE is used.

service-policy
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service policy that is defined in the DB2MQ.MQPOLICY table. A service policy specifies a set of quality-of-service options that are to be applied to this messaging operation. These options include message priority and message persistence. See *MQSeries Application Messaging Interface* for more details.

If *service-policy* is not specified or is the null value, DB2.DEFAULT.POLICY is used.

num-rows
An expression that returns a value that is a SMALLINT or INTEGER data type

whose value is a positive integer or zero. The value of the expression specifies the maximum number of messages to return.

If *num-rows* is not specified or if the value of the expression is zero, all available messages are returned.

The result of the function is a table with the format shown in the following table. All the columns in the table are nullable.

Table 85. Format of the resulting table for MQREADALLCLOB

Column name	Data type	Contains
MSG	CLOB(1M)	The contents of the MQSeries message
CORRELID	VARCHAR(24)	The correlation ID that is used to relate messages
TOPIC	VARCHAR(40)	The topic that the message was published with, if available
QNAME	VARCHAR(48)	The name of the queue from which the message was received
MSGID	CHAR(24)	The unique, MQSeries-assigned identifier for the message
MSGFORMAT	VARCHAR(8)	The format of the message, as defined by MQSeries

The CCSID of the result is the system CCSID that was in effect at the time that the MQSeries function was installed into DB2.

Example 1: Read all the messages from the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY).

```
SELECT *
  FROM SYSIBM.SYSDUMMY1 (MQREADALLCLOB()) AS T;
```

The messages and all the metadata are returned as a table.

Example 2: Read all the messages from the queue specified by the service MYSERVICE, using the default policy (DB2.DEFAULT.POLICY).

```
SELECT T.MSG, T.CORRELID
  FROM SYSIBM.SYSDUMMY1 (MQREADALLCLOB('MYSERVICE')) AS T;
```

Only the MSG and CORRELID columns are returned as a table.

Example 3: Read all the messages from the queue specified by the service MYSERVICE, using the default policy (DB2.DEFAULT.POLICY), with a correlation identifier of '1234'.

```
SELECT *
  FROM SYSIBM.SYSDUMMY1 (MQREADALLCLOB('MYSERVICE')) AS T
WHERE T.CORRELID = '1234';
```

All columns are returned.

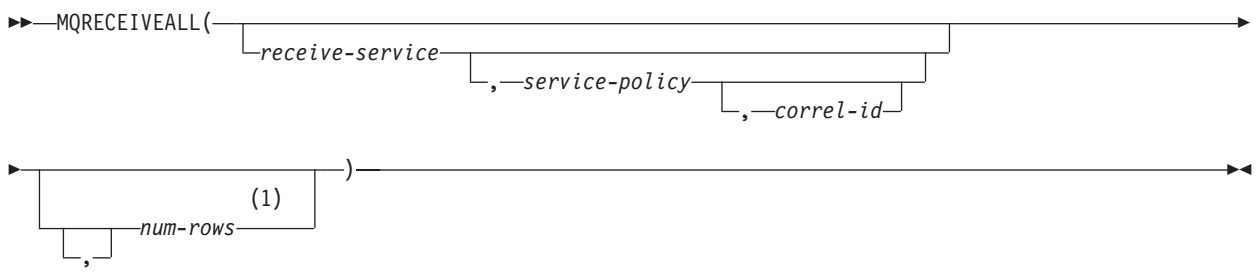
Example 4: Read the first 10 messages from the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY).

```
SELECT *
  FROM SYSIBM.SYSDUMMY1 (MQREADALLCLOB('10')) AS T;
```

All columns are returned.

MQRECEIVEALL

The MQRECEIVEALL function returns a table that contains the messages and message metadata from a specified MQSeries location and removes the messages from the queue.



Notes:

1 The comma is required before *num-rows* when any of the preceding arguments to the function are specified.

The schema is DB2MQ.

The MQRECEIVEALL function returns a table containing the messages and message meta-data from the MQSeries location that is specified by *receive-service*, using the quality-of-service policy that is defined in *service-policy*. Performing this operation removes the messages from the queue that is associated with *receive-service*.

receive-service
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service point that is defined in the DB2MQ.MQSERVICE table. A service point is a logical end-point from which a message is sent or received. A service point definition includes the name of the MQSeries queue manager and the name of the queue. See *MQSeries Application Messaging Interface* for more details.

If *receive-service* is not specified or is the null value, DB2.DEFAULT.SERVICE is used.

service-policy
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service policy that is defined in the DB2MQ.MQPOLICY table. A service policy specifies a set of quality-of-service options that are to be applied to this messaging operation. These options include message priority and message persistence. See *MQSeries Application Messaging Interface* for more details.

If *service-policy* is not specified or is the null value, DB2.DEFAULT.POLICY is used.

correl-id
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The expression must have an actual length that is no greater than 24 bytes. The value of the expression specifies the correlation identifier that is associated with this message. A correlation identifier is often specified in request-and-reply scenarios to associate requests with replies. Only those messages with a matching correlation identifier are returned.

A fixed length string with trailing blanks is considered a valid value. However, when the *correl-id* is specified on another request such as MQSEND, the *correl-id* must be specified the same to be recognized as a match. For example, specifying a value of 'test' for *correl-id* for this function does not match a *correl-id* value of 'test ' (with trailing blanks) specified earlier on an MQSEND request.

If *correl-id* is not specified, is an empty string, or is the null value, a correlation identifier is not used, and the message at the beginning of the queue is returned.

num-rows
An expression that returns a value that is a SMALLINT or INTEGER data type whose value is a positive integer or zero. The value of the expression specifies the maximum number of messages to return.

If *num-rows* is not specified or if the value of the expression is zero, all available messages are returned.

The result of the function is a table with the format shown in the following table. All of the columns are nullable.

Table 86. Format of resulting table for MQRECEIVEALL

Column name	Data type	Contains
MSG	VARCHAR(4000)	The contents of the MQSeries message
CORRELID	VARCHAR(24)	The correlation ID that is used to relate messages
TOPIC	VARCHAR(40)	The topic that the message was published with, if available
QNAME	VARCHAR(48)	The name of the queue from which the message was received
MSGID	CHAR(24)	The unique, MQSeries-assigned identifier for the message
MSGFORMAT	VARCHAR(8)	The format of the message, as defined by MQSeries

The CCSID of the result is the system CCSID that was in effect at the time that the MQSeries function was installed into DB2.

Example 1: Retrieve all the messages from the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY).

```
SELECT *
  FROM SYSIBM.SYSDUMMY1 (MQRECEIVEALL()) AS T;
```

The messages and all the metadata are returned as a table and the messages are removed from the queue.

Example 2: Retrieve all the messages from the the queue specified by the service MYSERVICE, using the default policy (DB2.DEFAULT.POLICY).

```
SELECT T.MSG, T.CORRELID
  FROM SYSIBM.SYSDUMMY1 (MQRECEIVEALL('MYSERVICE')) AS T;
```

Only the MSG and CORRELID columns are returned. The messages are removed from the queue.

Example 3: Retrieve all the messages from the beginning of the queue specified by the service MYSERVICE, using the policy MYPOLICY, with a correlation identifier of '1234'.

```
SELECT T.MSG, T.CORRELID
  FROM SYSIBM.SYSDUMMY1 (MQRECEIVEALL('MYSERVICE','MYPOLICY','1234')) AS T;
```

Only the MSG and CORRELID columns are returned. The messages are removed from the queue.

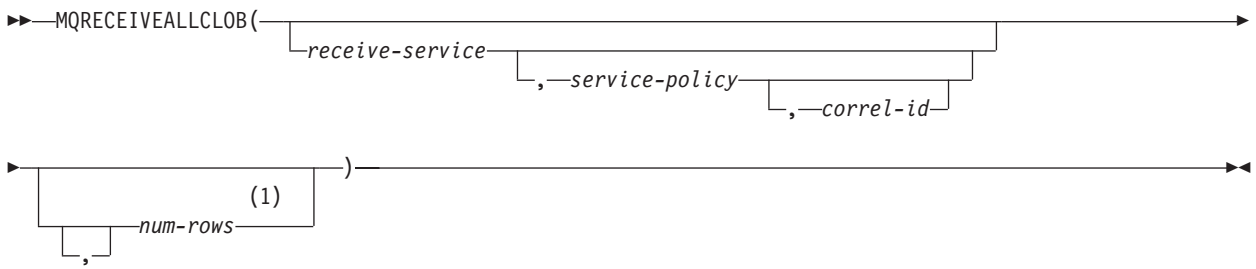
Example 4: Retrieve the first 10 messages from the beginning of the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY).

```
SELECT *
  FROM SYSIBM.SYSDUMMY1 (MQRECEIVEALL(10)) AS T;
```

All columns are returned. The messages are removed from the queue.

MQRECEIVEALLCLOB

The MQRECEIVEALLCLOB function returns a table that contains the messages and message metadata from a specified MQSeries location and removes the messages from the queue.



Notes:

1 The comma is required before *num-rows* when any of the preceding arguments to the function are specified.

The schema is DB2MQ.

The MQRECEIVEALLCLOB function returns a table containing the messages and message metadata from the MQSeries location that is specified by *receive-service*, using the quality-of-service policy that is defined in *service-policy*. Performing this operation removes the messages from the queue that is associated with *receive-service*.

receive-service
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service point that is defined in the DB2MQ.MQSERVICE table. A service point is a logical end-point from which a message is sent or received. A service point definition includes the name of the MQSeries queue manager and the name of the queue. See *MQSeries Application Messaging Interface* for more details.

If *receive-service* is not specified or is the null value, DB2.DEFAULT.SERVICE is used.

service-policy
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be an empty string or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression must refer to a service policy that is defined in the DB2MQ.MQPOLICY table. A service policy specifies a set of quality-of-service options that are to be applied to this messaging operation. These options include message priority and message persistence. See *MQSeries Application Messaging Interface* for more details.

If *service-policy* is not specified or is the null value, DB2.DEFAULT.POLICY is used.

correl-id
An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The expression must have an actual length that is no greater than 24 bytes. The value of the expression specifies the correlation identifier that is associated with this message. A correlation identifier is often specified in request-and-reply scenarios to associate requests with replies. Only those messages with a matching correlation identifier are returned.

A fixed length string with trailing blanks is considered a valid value. However, when the *correl-id* is specified on another request such as MQSEND, the *correl-id* must be specified the same to be recognized as a match. For example, specifying a value of 'test' for *correl-id* for this function does not match a *correl-id* value of 'test ' (with trailing blanks) specified earlier on an MQSEND request.

If *correl-id* is not specified, is an empty string, or is the null value, a correlation identifier is not used, and the message at the beginning of the queue is returned.

num-rows
An expression that returns a value that is a SMALLINT or INTEGER data type whose value is a positive integer or zero. The value of the expression specifies the maximum number of messages to return.

If *num-rows* is not specified or if the value of the expression is zero, all available messages are returned.

The result of the function is a table with the format shown in the following table. All of the columns are nullable.

Table 87. Format of resulting table for MQRECEIVEALLCLOB

Column name	Data type	Contains
MSG	CLOB(1M)	The contents of the MQSeries message
CORRELID	VARCHAR(24)	The correlation ID that is used to relate messages
TOPIC	VARCHAR(40)	The topic that the message was published with, if available
QNAME	VARCHAR(48)	The name of the queue from which the message was received
MSGID	CHAR(24)	The unique, MQSeries-assigned identifier for the message
MSGFORMAT	VARCHAR(8)	The format of the message, as defined by MQSeries

The CCSID of the result is the system CCSID that was in effect at the time that the MQSeries function was installed into DB2.

Example 1: Retrieve all the messages from the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY).

```
SELECT *
  FROM SYSIBM.SYSDUMMY1 (MQRECEIVEALLCLOB()) AS T;
```

The messages and all the metadata are returned as a table, and the messages are removed.

Example 2: Retrieve all the messages from the beginning of the queue specified by the service MYSERVICE, using the policy (DB2.DEFAULT.POLICY).

```
SELECT T.MSG, T.CORRELID
  FROM SYSIBM.SYSDUMMY1 (MQRECEIVEALLCLOB('MYSERVICE')) AS T;
```

Only the MSG and CORRELID columns are returned as a table, and the messages removed from the queue.

Example 3: Retrieve all the messages from the queue specified by the service MYSERVICE, using the policy "MYPOLICY", with a correlation identifier of '1234'.

```
SELECT *
  FROM SYSIBM.SYSDUMMY1 (MQRECEIVEALLCLOB('MYSERVICE','MYPOLICY','1234')) AS T;
```

All columns are returned, and the messages removed from the queue.

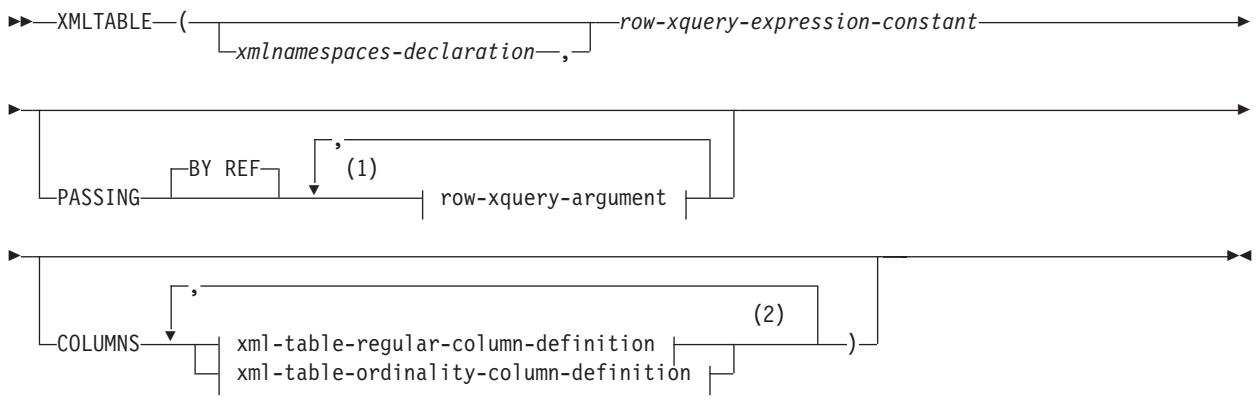
Example 4: Retrieve the first 10 messages from the beginning of the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY).

```
SELECT *
  FROM SYSIBM.SYSDUMMY1 (MQRECEIVEALLCLOB(10)) AS T;
```

All columns are returned, and the messages removed from the queue.

XMLTABLE

The XMLTABLE function returns a result table from the evaluation of XQuery expressions, possibly by using specified input arguments as XQuery variables. Each item in the result sequence of the row XQuery expression represents one row of the result table.

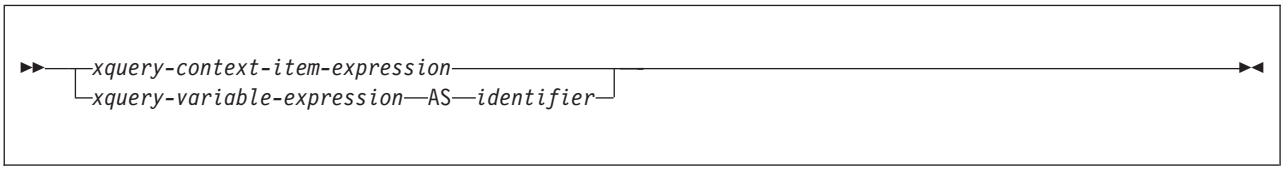


Notes:

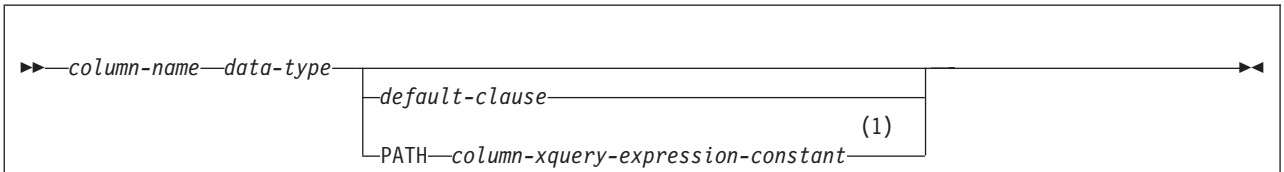
1 *xquery-context-item-expression* must not be specified more than one time.

2 The xml-table-ordinality-column-definition clause must not be specified more than one time.

row-xquery-argument



xml-table-regular-column-definition



Notes:

1 Neither the *default-clause* or the **PATH** clause can be specified more than one time.

xml-table-ordinality-column-definition

The schema is SYSIBM.

The function name cannot be specified as a qualified name.

xmlnamespaces-declaration
Specifies one or more XML namespace declarations, using the XMLNAMESPACES function, that become part of the static context of the *row-xquery-expression-constant* and the *column-xquery-expression-constant*. The set of statically known namespaces for XQuery expressions which are arguments of XMLTABLE is the combination of the pre-established set of statically known namespaces and the namespace declarations specified in this clause. The XQuery prolog within an XQuery expression can override these namespaces.

If *xmlnamespaces-declaration* is not specified, only the pre-established set of statically known namespaces apply to the XQuery expressions.

row-xquery-expression-constant
Specifies an SQL character string constant that is interpreted as an XQuery expression using supported XQuery language syntax. *row-xquery-expression-constant* cannot be an XQuery updating expression. This expression determines the number of rows in the result table. The expression is evaluated using the optional set of input XML values that is specified in *row-xquery-argument*, and returns an output XQuery sequence where one row is generated for each item in the sequence. If the sequence is empty, the result of XMLTABLE is an empty table. *row-xquery-expression-constant* must not contain an empty string or a string of all blanks.

PASSING
Specifies input values and the manner in which these values are passed to *row-xquery-expression-constant*.

BY REF
Specifies that any XML input arguments are, by default, passed by reference. When XML values are passed by reference, the XQuery evaluation uses the input node trees, if any exist, directly from the specified input expressions and preserves all properties, including the original node identities and document order.

This clause has no impact on how non-XML values are passed. The non-XML values create a new copy of the value during the cast to XML.

row-xquery-argument
Specifies an argument that is to be passed to the XQuery expression specified by *row-xquery-expression-constant*. *row-xquery-argument* is an SQL expression that returns a value that is not a ROWID, LOB, DATE, TIME, TIMESTAMP, BINARY, VARBINARY, REAL, DECFLOAT, or character string with FOR BIT DATA attribute.

How *row-xquery-argument* is used in the XQuery expression depends on whether the argument is specified as an *xquery-context-item-expression* or an *xquery-variable-expression*.

If the data type of *row-xquery-argument* is not XML, the result of the expression for the argument is implicitly cast to XML. A null value is converted to an XML empty sequence if the argument is *xquery-variable-expression*.

row-xquery-argument must not contain NEXT VALUE or PREVIOUS VALUE expressions or OLAP specifications.

xquery-context-item-expression
An expression that returns a value that is XML, integer, decimal, or a character or graphic string that is not a LOB. *xquery-context-item-expression* must not be a character string that is bit data.

xquery-context-item-expression specifies the initial context item for the *row-xquery-expression*. The value of the initial context item is the result of *xquery-context-item-expression* cast to XML. *xquery-context-item-expression* must not be specified more than one time.

xquery-variable-expression
Specifies an SQL expression whose value is available to the XQuery expression specified by *row-xquery-expression-constant* during execution. The expression must returns a value that is XML, integer, decimal, or a character or graphic string that is not a LOB.

xquery-variable-expression specifies an argument that will be passed to *row-xquery-expression-constant* as an XQuery variable. If *xquery-variable-expression* is a null value, the XQuery variable is set to an XML empty sequence. The scope of the XQuery variables that are created from the PASSING clause is the XQuery expression specified by *row-xquery-expression-constant*.

AS *identifier*
Specifies that the value generated by *xquery-variable-expression* will be passed to *row-xquery-expression-constant* as an XQuery variable. The variable name will be *identifier*. The leading dollar sign ($) that precedes variable names in the XQuery language is not included in *identifier*. The identifier must not be greater than 128 bytes in length. Two arguments within the same PASSING clause cannot use the same identifier.

COLUMNS
Specifies the output columns of the result table including the column name, data type, and how the column value is computed for each row. If this clause is not specified, a single unnamed column of data type XML is returned, with the value based on the sequence item from evaluating the XQuery expression in the *row-xquery-expression-constant* (equivalent to specifying PATH '.'). To reference the result column, a *column-name* must be specified in the *correlation-clause* following the function.

xml-table-regular-column-definition
Specifies one output column of the result table including the column name, data type, and an XQuery expression to extract the value from the sequence item for the row.

column-name
Specifies the name of the column in the result table. The name cannot be qualified and the same name cannot be used for more than one column of the table.

data-type
Specifies the data type of the column. See CREATE TABLE for the

syntax and a description of types available. A *data-type* can be used in XMLTABLE if there is a supported XMLCAST from the XML data type to the specified *data-type*.

default-clause
Specifies a default value for the column. See CREATE TABLE for the syntax and a description of the *default-clause*. For XMLTABLE result columns, the default is applied when the processing of the XQuery expression contained in *column-xquery-expression-constant* returns an empty sequence. This default value will not be inherited by declared global temporary tables even when the INCLUDING COLUMN DEFAULTS clause is specified in the definition of the declared global temporary table.

PATH *column-xquery-expression-constant*
Specifies an SQL character string constant that is interpreted as an XQuery expression using supported XQuery language syntax. The *column-xquery-expression-constant* specifies an XQuery expression that determines the column value with respect to an item that is the result of evaluating the XQuery expression in *row-xquery-expression-constant*. Given an item from the result of processing the *row-xquery-expression-constant* as the externally provided context item, the *column-xquery-expression-constant* is evaluated and returns an output sequence. The column value is determined based on this output sequence as follows.

- If the output sequence contains zero items, the *default-clause* provides the value of the column.
- If an empty sequence is returned and no *default-clause* was specified, a null value is assigned to the column.
- If a non-empty sequence is returned, the value is cast to the *data-type* specified for the column using the XMLCAST expression. An error could be returned from processing this XMLCAST.

The value for *column-xquery-expression-constant* must not be an empty string or a string of all blanks. If this clause is not specified, the default XQuery expression is simply the *column-name*.

xml-table-ordinality-column-definition
Specifies the ordinality column of the result table.

column-name
Specifies the name of the column in the result table. The name cannot be qualified and the same name cannot be used for more than one column of the table.

FOR ORDINALITY
Specifies that *column-name* is the ordinality column of the result table. The data type of this column is BIGINT. The value of this column in the result table is the sequential number of the item for the row in the resulting sequence from evaluating the XQuery expression in *row-xquery-expression-constant*.

The result of the function is a table. The encoding scheme of the table is Unicode. If the evaluation of any of the XQuery expressions results in an error, the XMLTABLE function returns the XQuery error.

Example: List as a table result the purchase order items for orders with a status of 'NEW':

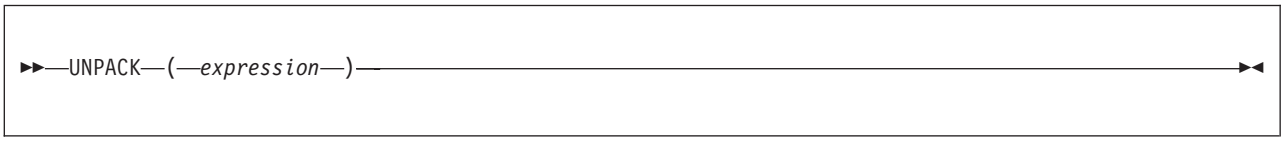
```
SELECT U."PO ID", U."Part #", U."Product Name",
     U."Quantity", U."Price", U."Order Date"
  FROM PURCHASEORDER P,
     XMLTABLE(XMLNAMESPACES('http://podemo.org' AS "pod"),
              '$po/PurchaseOrder/itemlist/item' PASSING P.PORDER as "po"
              COLUMNS "PO ID"         INTEGER       PATH '../../@POid',
                      "Part #"        CHAR(6)       PATH 'product/@pid',
                      "Product Name"  CHAR(50)      PATH 'product/pod:name',
                      "Quantity"      INTEGER       PATH 'quantity',
                      "Price"         DECIMAL(9,2)  PATH 'product/pod:price',
                      "Order Date"    TIMESTAMP     PATH '../../dateTime'
              ) AS U
   WHERE P.STATUS = 'NEW'
```

Row functions

A row function can be used only in contexts that are specifically described for the function.

UNPACK

The UNPACK function returns a row of values that are derived from unpacking the input binary string. It is used to unpack a string that was encoded according to the PACK function.



The schema is SYSIBM.

expression
An expression that returns the string value to be unpacked. The *expression* must be a binary string that is not a BLOB and that is not null. The format of the binary string must match the one that is produced by the PACK function.

The UNPACK function can only be specified in the SELECT list and the SET clause of the UPDATE statement.

The result of the function is a row of fields corresponding to the data elements that were encoded in the input packed string. The result is not null.

Example 1: Assume that a user-defined function named myUDF returns a VARBINARY result. The body of the function includes the following invocation of the PACK function to pack some data into a binary string:

```
SET :udf_result = PACK(CCSID 1208, 'Alina', DATE'1977-08-01',
                       DOUBLE(0.5));
```

The following SELECT statement unpacks the result of the myUDF function and returns a row of individual column values:

```
SELECT UNPACK(myUDF(C1)).* AS(Name VARCHAR(40) CCSID UNICODE,
                              DOB DATE,
                              Score DOUBLE)
  FROM T1;
```

The use of ".*" indicates that the result of the UNPACK function should be flattened into a list of result column values. When the UNPACK function is used in a select clause, an AS clause is specified to provide the names and data types for the resulting values.

Example 2: Assume that a user-defined function UDF_SCORE returns a VARBINARY result. The PACK function is invoked to return a binary string in which the column values of table T1 are encoded and packed. The UNPACK function returns the individual data values for a row with column names ID, SCORE, and CONF:

```
SELECT T1.C1, T1.C2, T1.C3, T1.C4,
       UNPACK( UDF_SCORE( PACK(CCSID 1208, T1.C1, T1.C2, T1.C3)) ).*
              AS (ID INT, SCORE DOUBLE, CONF DOUBLE)
  FROM T1;
```

Related reference:

"PACK" on page 562

"select-clause" on page 765

"unpacked-row" on page 771

Chapter 4. Queries

A *query* specifies a result table or an intermediate table. A query is a component of certain SQL statements. A query can have one of three forms.

A "subselect" on page 764
A "fullselect" on page 811
A "select-statement" on page 819

A subselect is a subset of a fullselect, and a fullselect is a subset of a *select-statement*.

Restriction: For all three forms of a query, you cannot reference both a system-period temporal table and an archive-enabled table in the same query.

"Authorization" on page 762 describes the privilege set that is required to use any form of a query.

Another SQL statement that can be used to retrieve at most a single row is described in "SELECT INTO" on page 1866. SELECT INTO is not a subselect, fullselect, or a *select-statement*.

Related concepts:

Types of tables (Introduction to DB2 for z/OS)

Temporal tables (DB2 Administration Guide)

Archive-enabled tables and archive tables (Introduction to DB2 for z/OS)

Authorization

For any form of a query, the privilege set that is defined below must include one of the following:

- For each table or view identified in the statement, the privilege set must include one of the following:
  - Ownership of the table or view
  - The SELECT privilege on the table or view
  - DBADM authority for the database (tables only)

  If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.
- SYSADM authority
- SYSCTRL authority (catalog tables only)
- DATAACCESS authority

If a query includes a user-defined function, the privileges that are held by the authorization ID of the statement must include at least one of the following:

- For each user-defined function that is identified in the statement, one of the following:
  - The EXECUTE privilege on the function
  - Ownership of the function
- SYSADM authority
- DATAACCESS authority

If the *select-statement* is part of a DECLARE CURSOR statement, the privilege set is the privileges that are held by the authorization ID of the owner of the plan or package.

If the *select-statement* contains an SQL data change statement, the privilege set must include the SELECT privilege and the appropriate privileges for the SQL data change statement (insert, update, or delete privileges) on the target table or view.

If the *select-statement* references a table that contain an active row or column access control, and row permissions or column masks are defined for the table, the authorization ID or role of the statement does not need authority to reference objects that are specified in the definitions of those row permissions or column masks.

For dynamically prepared statements, the privilege set depends on the dynamic SQL statement behavior, which is specified by option DYNAMICRULES:

Run behavior
: The privilege set is the union of the privilege sets that are held by each authorization ID of the process.

Bind behavior
: The privilege set is the privileges that are held by the authorization ID of the owner of the plan or package.

Define behavior
: The privilege set is the privileges that are held by the authorization ID of the owner of the stored procedure or user-defined function.

Invoke behavior
The privilege set is the privileges that are held by the authorization ID of the invoker of the stored procedure or user-defined function.

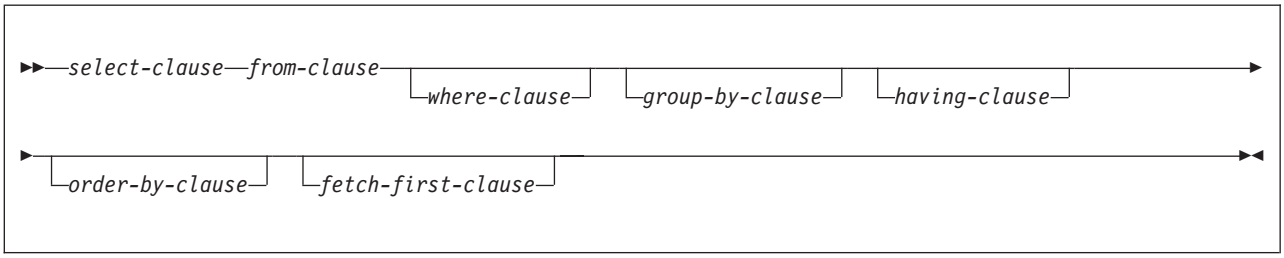
For a list of the DYNAMICRULES values that specify run, bind, define, or invoke behavior, see Table 6 on page 75.

When any form of a query is used as a component of another statement, the authorization rules that apply to the query are specified in the description of that statement. For example, see "CREATE VIEW" on page 1527 for the authorization rules that apply to the subselect component of CREATE VIEW.

If your installation uses the access control authorization exit (DSNX@XAC), that exit might be controlling the authorization rules instead of the rules that are listed here.

subselect

The *subselect* is a component of the fullselect. A subselect specifies a result table that is derived from the tables or views that are identified in the FROM clause.



The derivation of the result table can be described as a sequence of operations in which the result of each operation is input for the next. (This is only a way of describing the subselect. The method that is used to perform the derivation might be quite different from this description. If portions of the subselect do not actually need to be executed for the correct result to be obtained, they might not be executed.)

When a subselect directly or indirectly references a table for which row or column access control is enforced, the rules that are defined in the row permissions or column masks affect how the rows in the result table are derived. Typically those rules are based on the authorization ID or role of the process.

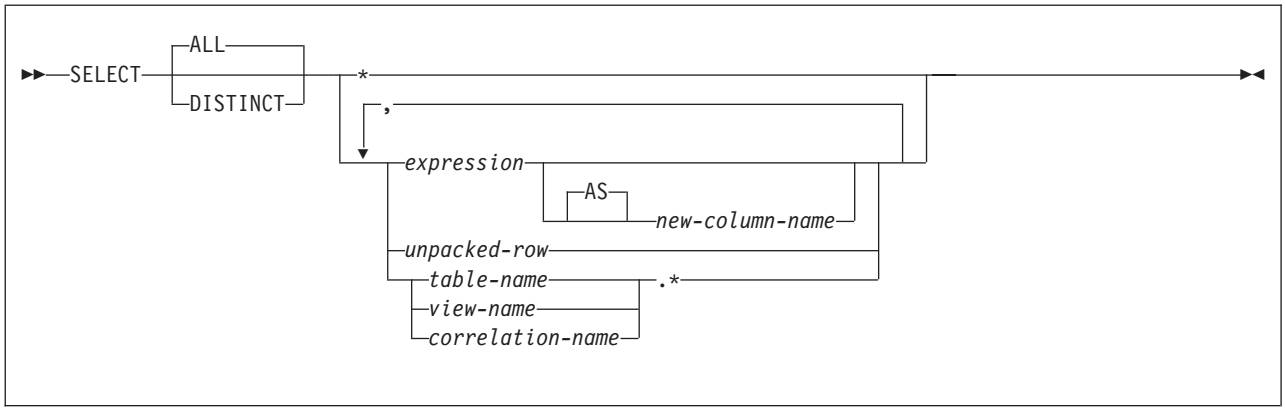
A *scalar-subselect* is a subselect, enclosed in parentheses, that returns a single result row and a single result column. If the result of the subselect is no rows, the null value is returned. An error is returned if the result contains more than one row.

The clauses of the subselect are processed in the following sequence:
1. FROM clause
2. WHERE clause
3. GROUP BY clause
4. HAVING clause
5. SELECT clause
6. ORDER BY clause
7. FETCH FIRST clause

select-clause

The SELECT clause specifies the columns of the final result table. The column values are produced by the application of the *select list* to R. The select list is a list of names and expressions specified in the SELECT clause, and R is the result of the previous operation of the subselect. For example, if SELECT, FROM, and WHERE are the only clauses specified, then R is the result of that WHERE clause.

select-clause



ALL

Retains all rows of the final result table and does not eliminate redundant duplicates. This is the default.

DISTINCT

Eliminates all but one of each set of duplicate rows of the final result table.

Two rows are duplicates of one another only if each value in the first row is equal to the corresponding value in the second row. For determining duplicate rows, two null values are considered equal.

When SELECT DISTINCT is specified, no column or expression in the implicit or explicit list can return a value that is a LOB or XML data type. When a column or expression in the list returns a value that is a DECFLOAT data type and multiple bit representations of the same number exists in the intermediate result, the value that is returned is unpredictable. See "Numeric comparisons" on page 134 for additional information.

Column access controls do not affect the operation of SELECT DISTINCT. The elimination of duplicated rows is based on the original column values, not the masked values. However, after the application of column masks, the masked values in the final result table might not reflect the uniqueness that is enforced by SELECT DISTINCT.

If a column mask is applied to a column that directly or indirectly derives the result of SELECT DISTINCT, SELECT DISTINCT can return a result that is not deterministic. The following conditions are a few examples of when a result that is not deterministic might be returned:

- The definition of the column mask references other columns of the table to which the column mask is applied.
- The column is referenced in the argument of a built-in scalar function, such as COALESCE, IFNULL, NULLIF, MAX, MIN, LOCATE, TOTALORDER, etc.
- The column is referenced in the argument of an aggregation function.

- The column is embedded in an expression and the expression contains a function that is not deterministic or has an external action.

For compatibility with other SQL implementations, UNIQUE can be specified as a synonym for DISTINCT.

Select list notation:

***** Represents a list of columns of table R, excluding any columns that are defined as implicitly hidden. The list of names is established when the statement containing the SELECT clause is prepared. Therefore, * does not identify any columns that have been added to a table after the statement has been prepared.

A column that is defined as implicitly hidden can be explicitly referenced in the select list.

* cannot be used in the definition of a row permission or a column mask.

expression
Specifies the values of a result column. Each *column-name* in the expression must unambiguously identify a column of the intermediate result table.

AS *new-column-name*
Names or renames the result column. The name must not be qualified and does not have to be unique. *new-column-name* is an SQL identifier of 128 UTF-8 bytes or less.

*name.**
Represents a list of columns of *name*, excluding any columns that are defined as implicitly hidden, in the order the columns are produced by the FROM clause. *name* can be a table name, view name, or correlation name, and must designate an exposed table, view, or correlation name in the FROM clause that immediately follows the SELECT clause. The first name in the list identifies the first column of the table or view, the second name in the list identifies the second column of the table or view, and so on.

The list of names is established when the statement that contains the SELECT clause is prepared. Therefore, * does not identify any columns that have been added to a table after the statement has been prepared.

*name.** cannot be used in the definition of a row permission or a column mask.

SQL statements can be implicitly or explicitly prepared again. The effect of another prepare on statements that include * or *name.** is that the list of names is re-established. Therefore, the number of columns returned by the statement might change.

The number of columns in the result of SELECT is the same as the number of expressions in the operational form of the select list (that is, the list established at the time the statement is prepared), and cannot exceed 750. The result of a subquery must be a single column unless the subquery is used in an EXISTS predicate.

Notes:

If the FROM clause contains a MERGE statement:
The SELECT list must not implicitly or explicitly refer to a column that has a LOB data type, a ROWID data type (or a distinct type that is based on a LOB, or ROWID), or an XML data type.

Implicitly hidden ROWID columns in the select list:
The result for SELECT * does not include any implicitly hidden ROWID columns. To be included in the result, implicitly hidden ROWID columns must be explicitly specified in the select list.

VARBINARY data:
If the identified table has an index on a VARBINARY column or a column that is a distinct type that is based on VARBINARY data type, that index column cannot specify the DESC attribute. To query the identified table, either drop the index or alter the data type of the column to BINARY and then rebuild the index.

Applying the select list:
Some of the results of applying the select list to R depend on whether GROUP BY or HAVING is used. The following three lists describe the results.

IF neither GROUP BY nor HAVING is used:

- The select list can include aggregate functions only if it includes other aggregate functions, constants, or expressions that only involve constants.
- If the select list does not include aggregate functions, it is applied to each row of R and the result contains as many rows as there are rows in R.
- If the select list includes aggregate functions, R is the source of the arguments of the functions and the result of applying the select list is one row, even when R has no rows.
- If a column mask is used to mask the values in the final result table, and the select list includes aggregate functions, the definition of the column mask must not reference the following:
  - A scalar fullselect
  - An aggregate function

If HAVING is used and GROUP BY is not used:
Each expression or *column-name* in an expression in the select list must be specified within an aggregate function. Constants or expressions that involve only constants can also be in the select list.

If a column mask is used to mask the values in the final result table, the definition of the column mask must not reference the following:

- A scalar fullselect
- An aggregate function

If GROUP BY is used:

- Each expression in the select list must use one or more grouping expressions. Or, each expression or *column-name* in an expression must:
  - Unambiguously identify a grouping column of R.
  - Be specified within an aggregate function.
  - Be a correlated reference. (A column-name is a correlated reference if it identifies a column of a table or view identified in an outer subselect.)
- If an expression in the select list is a scalar fullselect, a correlated reference from the scalar fullselect to a group R must either identify a grouping column or be contained within an aggregate

function. For example, the following query fails because the correlated reference T1.C1 || T1.C2 in the select list of the scalar fullselect does not match a grouping column from the outer subselect. (Matching the grouping expression T1.C1 || T1.C2 is not supported.)

```
SELECT MAX(T1.C2) AS X1,
     (SELECT T1.C1 || T1.C2 FROM T2 GROUP BY T2.C1) AS  Y1
  FROM T1
  GROUP BY  T1.C1, T1.C1 || T1.C2;
```

- You cannot use GROUP BY with a name defined using the AS clause unless the name is defined in a nested table expression. Example 6 demonstrates the valid use of AS and GROUP BY in a SELECT statement.

In either case, the *n*th column of the result contains the values specified by applying the *n*th expression in the operational form of the select list.

If a column mask is used to mask the column values in the final result table, a column for which the column mask is applied must satisfy one of the following conditions:

- The column must be specified in an aggregate function and the definition of the column mask must not reference the following:
  - A scalar fullselect
  - An aggregate function
- The column must identify a *column-name* in the GROUP BY clause and the column must not be referenced in an *expression* in the GROUP BY clause. In addition, any columns of the same table as the column for which the column mask is applied and are referenced in the definition of the column mask must be identified with a *column-name* in the GROUP BY clause. These columns must not be referenced in an *expression* in the GROUP BY clause.
- A column of a non-base tables in the select list must be specified in an aggregate function if a column mask is used to mask the column values in the final result table, and the column of a non-base table maps directly or indirectly to a column name or to an expression in a materialized table expression or view to the table where the column mask is applied.

Effect of column masks on result columns:

When column masks are enabled, they determine the values in the final result table of an outermost select list. When a column mask is enabled for a column, if the column appears in the outermost select list (either implicitly or explicitly), the column mask is applied to the column to produce the values for the final result table. If the column itself does not appear in the outermost select list, but is included in the output (for example, it appears in a materialized table expression or a view), the masked value is included in the result table of the table expression or view so that it can be used in the final result table.

The enabled column masks do not interfere with the operations of other clauses within the statement, such as the WHERE, GROUP BY, HAVING, SELECT DISTINCT, and ORDER BY clauses.

The rows that are returned in the final result table remain the same, except that the values in the result rows might be masked. As such, if a column

with masked values also appears in an ORDER BY clause with a *sort-key* expression, the order is based on the original column values (the masked values in the final result table might not reflect that order). Similarly, the masked values might not reflect the uniqueness enforced by a SELECT DISTINCT. If the masked column is embedded in an expression, the result of the expression might be different because the column mask is applied to the column before the expression is evaluated. For example, a column mask on column SSN can change the result of the function COUNT(DISTINCT SSN) because the DISTINCT operation is performed on the masked values. However, if the expression in the query is the same as the expression that is used to mask the column value in the definition of the column mask, the result of the expression might remain unchanged. For example, the expression in the query is `'XXX-XX-' || SUBSTR(SSN, 8, 4)` and the same expression is used in the column mask definition. In this particular example, the expression in the query can be replaced with column SSN to avoid the same expression being evaluated twice.

If a CASE expression appears in the outermost select list, column masks are not applied to the *search-condition* of the WHEN clause.

When the definition of a column mask is applied to an SQL statement to mask column values in the final result table, the semantics of the column mask might conflict with certain SQL semantics in the statement. In these situations, the combination of the statement and the column mask might return an error.

See "ALTER TABLE" on page 984 for more information about the application of enabled column masks.

Null attributes of result columns:

Result columns allow null values if they are derived from one of the following:

- Any aggregate function except COUNT or COUNT_BIG
- A column that allows null values
- A view column in an outer select list that is derived from an arithmetic expression
- An arithmetic expression in an outer select list
- An arithmetic expression that allows nulls
- A scalar function or string expression that allows null values
- A host variable that has an indicator variable, an SQL parameter or variable, a global variable, or in the case of Java, a host variable or expression whose type is able to represent a Java null value
- A result of a set operator if at least one of the corresponding items in the select list is nullable

Names of result columns:

In the following cases a result column is considered a named column:

- If the AS clause is specified, the name of the result column is the name specified on the AS clause.
- If the AS clause is not specified and a column list is specified in the correlation clause, the name of the result column is the corresponding name in the correlation column list.
- If neither an AS clause nor a column list in the correlation clause is specified and the result column is derived only from a single column (without any functions or operators), the result column name is the unqualified name of that column.

- If neither an AS clause nor a column list in the correlation clause is specified and the result column is derived only from a single SQL variable, global variable, or SQL parameter (without any functions or operators), the result column name is the unqualified name of that SQL variable, global variable, or SQL parameter.

In all other cases, a result column is an unnamed column.

Names of result columns, SQL variables, and global variables are placed into the SQL descriptor area (SQLDA) when the DESCRIBE statement is executed. This allows an interactive SQL processor such as SPUFI, the command line processor, or DB2 QMF to use the column names when displaying the results. The names in the SQLDA include those specified by the AS clause.

Data types of result columns:

Each column of the result of SELECT acquires a data type from the expression from which it is derived. The following table shows the data types of result columns.

Table 88. Data types of result columns

When the expression is...	The data type of the result column is...
The name of any numeric column	The same as the data type of the column, with the same precision and scale for decimal columns.
An integer constant	INTEGER.
A decimal or floating-point constant	The same as the data type of the constant, with the same precision and scale for decimal constants. For floating-point constants, the data type is DOUBLE PRECISION.
A decimal floating point constant	DECFLOAT(34)
The name of any numeric host variable	The same as the data type of the variable, with the same precision and scale for decimal variables. The result is decimal if the data type of the host variable is not an SQL data type; for example, DISPLAY SIGN LEADING SEPARATE in COBOL.
An arithmetic or string expression	The same as the data type of the result, with the same precision and scale for decimal results as described in "Expressions" on page 240.
Any function	The data type of the result of the function. For a built-in function, see Chapter 3, "Functions," on page 337 to determine the data type of the result. For a user-defined function, the data type of the result is what was defined in the CREATE FUNCTION statement for the function.
The name of any string column	The same as the data type of the column, with the same length attribute.
The name of any string host variable	The same as the data type of the variable, with a length attribute equal to the length of the variable. The result is a varying-length character string if the data type of the host variable is not an SQL data type; for example, a NUL-terminated string in C.
A character string constant of length *n*	VARCHAR(*n*).
A binary string constant of length *n*	VARBINARY(*n*)
A graphic string constant of length *n*	VARGRAPHIC(*n*).
The name of a datetime column	The same as the data type of the column.
The name of a ROWID column	Row ID.
The name of a distinct type column	The same as the distinct type of the column, with the same length, precision, and scale attributes, if any.

For information about the CCSID of the result column, see "Rules for result data types" on page 144.

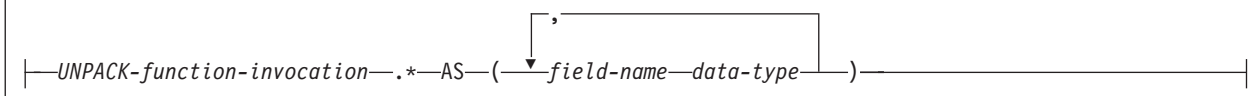
Related reference:

"Examples of subselects" on page 805

unpacked-row

An *unpacked-row* specifies a row that is the result of an invocation of the UNPACK built-in function.

unpacked-row:



UNPACK-function-invocation
Specifies an invocation of the UNPACK built-in function. The number of specified *field-names* and *field-types* must be the same as the number of fields that are returned by the UNPACK function invocation.

field-name
Names the field that is returned from the UNPACK function. A name must not be qualified, and it does not have to be unique.

data-type
Specifies the built-in data type of the field. The specified data type, length, and CCSID must correspond to the data type, length, and CCSID of the data when the argument was initially encoded with the PACK function. The following table provides the supported data type mappings from the packed string data:

Table 89. Data type mappings from packed string data

Data type of an encoded value in the packed string for UNPACK	**Data type specified for UNPACK**
SMALLINT	SMALLINT, INTEGER, BIGINT
INTEGER	INTEGER, BIGINT
BIGINT	BIGINT
decimal (*p*,*s*)[1]	decimal(*p'*, *s'*) if *s'*< *s*, *s*-*s'* digits are truncated. An error occurs if there are more than *p'*-*s'* significant digits.
real[2] or double[3]	double
CHAR(*n*) or VARCHAR(*n*)	CHAR(*m*), VARCHAR(*m*) If *m* < *n* and any of the *n*-*m* characters is not a blank, an error occurs. Otherwise, the n-m blanks are truncated. If *m* > *n* and the specified data type is CHAR, *m*-*n* blanks are appended.

Table 89. Data type mappings from packed string data (continued)

Data type of an encoded value in the packed string for UNPACK	**Data type specified for UNPACK**
BINARY(*n*) or VARBINARY(*n*)	BINARY(*m*), VARBINARY(*m*) If *m* < *n*, an error occurs. If *m* > *n* and the UNPACK target is BINARY, *m-n* X'00' bytes are appended.
DATE	DATE
TIME	TIME
TIMESTAMP(*p*) WITHOUT TIME ZONE	TIMESTAMP(*p'*) WITHOUT TIME ZONE. If *p'* > *p*, *p'-p* zeros are appended. If *p'* < *p*, *p-p'* digits are truncated.
TIMESTAMP(*p*) WITH TIME ZONE	TIMESTAMP(*p'*) WITH TIME ZONE. If *p'* > *p*, *p'-p* zeros are appended. If *p'* < *p*, *p-p'* digits are truncated.
Note: The data types in lower case are defined as follows: 1. **decimal** = DECIMAL(*p,s*) or NUMERIC(*p,s*) 2. **real** = REAL or FLOAT(*n*) where *n* is the specification for a single precision floating point 3. **double** = DOUBLE, DOUBLE PRECISION, FLOAT or FLOAT(*n*) where *n* is the specification for a double precision floating point The synonyms for the data types, in either long or short form, are considered the same as those that are listed.	

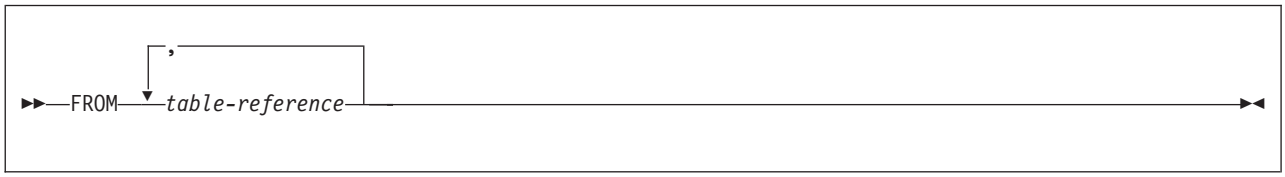
Related reference:

"select-clause" on page 765

from-clause

The FROM clause specifies an intermediate result table.

from-clause



If only one *table-reference* is specified, the intermediate result table is simply the result of that *table-reference*. If more than one *table-reference* is specified, the intermediate result table consists of all possible combinations of the rows of the result of each specified *table-reference*.

Each row of the result is a row from the result of the first *table-reference* concatenated with a row from the result of the second *table-reference*, concatenated with a row from the result of the third *table-reference*, and so on. The number of rows in the result is the product of the number of rows in the result of each *table-reference*. Thus, if the result of any *table-reference* is empty, the result is empty.

If *table-reference* has row access controls enforced, *table-reference* has at least one row permission: the default row permission. When there are multiple row permissions defined for a *table-reference*, a row access control search condition is derived by applying the logical OR operator to the search condition in each enabled permission. This derived search condition acts as a filter to the *table-reference* to determine the result table of the *table-reference* that is accessible to the authorization ID or role of the subselect.

If a *table-reference* contains a security label column, DB2 compares the security label of the user to the security label of each row. Results are returned according to the following rules:

- If the security label of the user dominates the security label of the row, DB2 returns the row.
- If the security label of the user does not dominate the security label of the row, DB2 does not return the data from that row, and DB2 does not generate an error report.

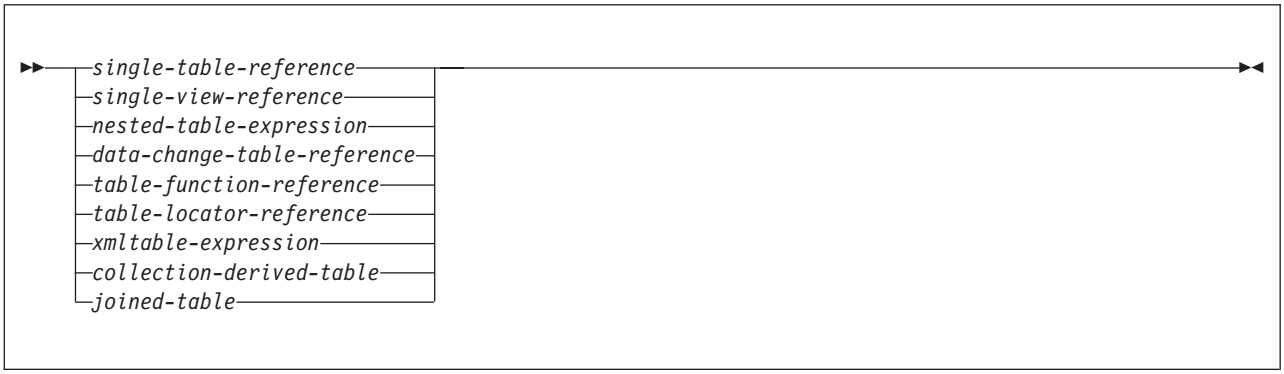
Related reference:

"Examples of subselects" on page 805

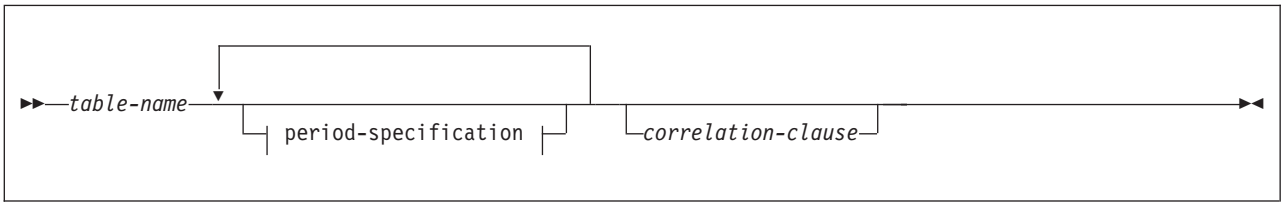
table-reference

A *table-reference* specifies a result table as either a table or view, or an intermediate table.

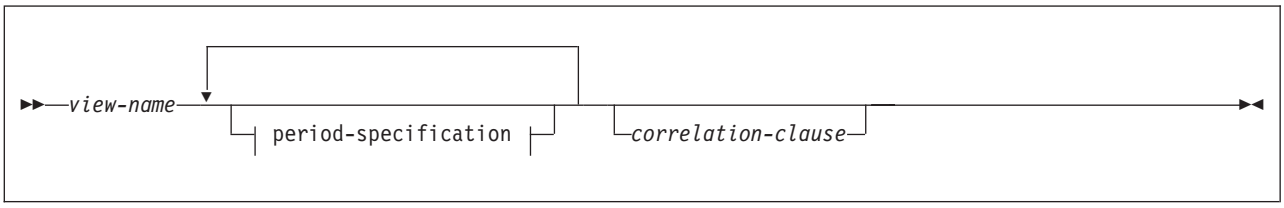
table-reference:



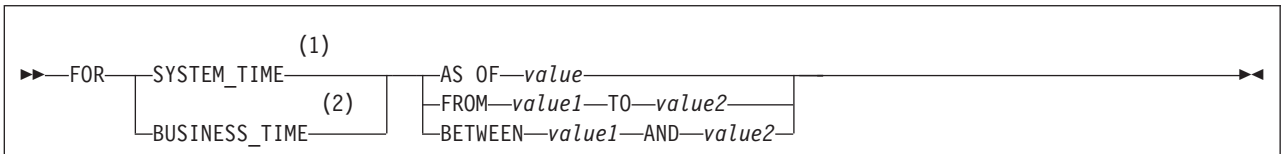
single-table-reference:



single-view-reference:



period-specification:

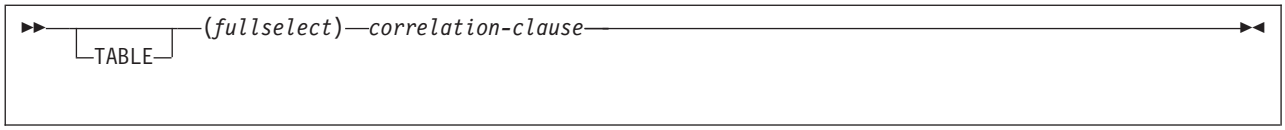


Notes:

1 AS OF TIMESTAMP can be specified in place of FOR SYSTEM_TIME AS OF.

2 SYSTEM_TIME and BUSINESS_TIME cannot be specified more than one time per table.

nested-table-expression:



data-change-table-reference:

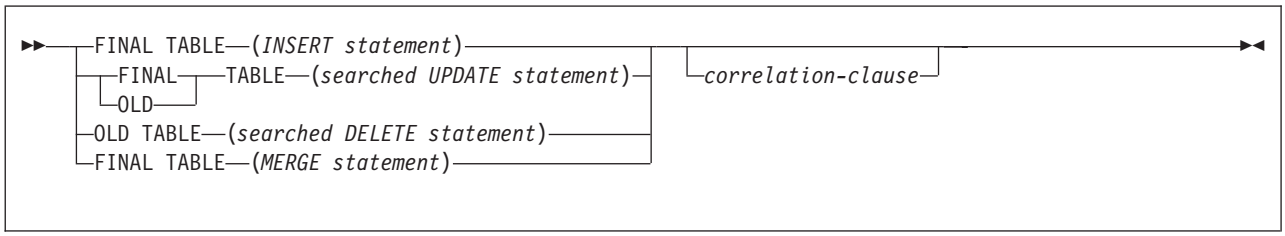
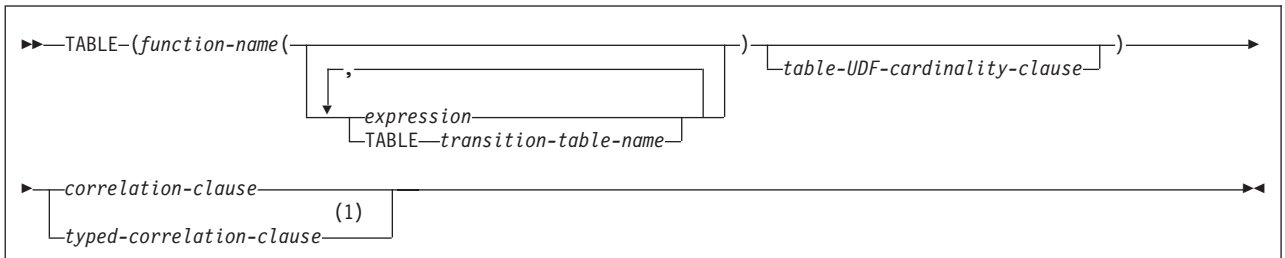


table-function-reference:



Notes:

1 The *typed-correlation-clause* is required for generic table functions. This clause cannot be specified for any other table functions.

table-UDF-cardinality-clause:

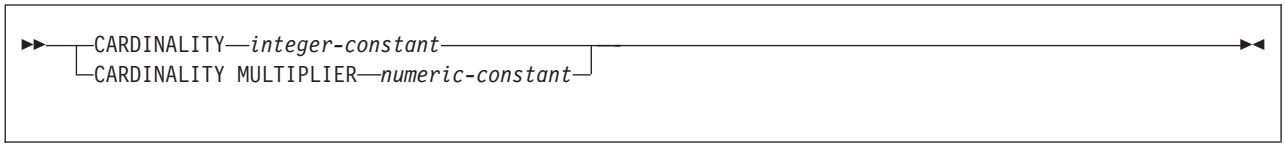
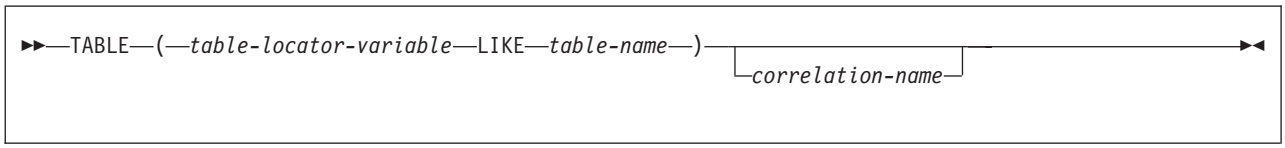
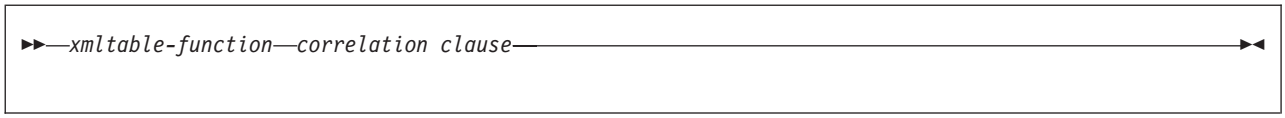


table-locator-reference:



xmltable-expression:



A *table-reference* specifies an intermediate result table.

- If a *single-table-reference* is specified and it is not an archive-enabled table or a temporal table, the intermediate result table is the specified table. If a *period-specification* is also specified, the intermediate result table consists of the rows of the temporal table where the period matches the specification.
- If a *single-table-reference* is specified and it is an archive-enabled table, the setting of the SYSIBMADM.GET_ARCHIVE global variable and the ARCHIVESENSITIVE bind option determine the contents of the intermediate result table. If the global variable is set to Y and the bind option is set to YES, the intermediate result table includes the rows in the associated archive table. Otherwise, the intermediate result table does not include rows in the associated archive table.
- If a *single-view-reference* is specified without a *period-specification*, the intermediate result table is that view. If a *period-specification* is specified, temporal table references in the view consider only the rows where the period matches the specification.
- If a *nested-table-expression* is specified, the result table is the result of the specified fullselect. The columns of the result do not need unique names, but a column with a non-unique name cannot be explicitly referenced.
- If a *data-change-table-reference* is specified, the intermediate result table is the set of rows that are directly affected by the data change statement.
- If a *table-function-reference* is specified, the intermediate result table is the set of rows that are returned by the table function.
- If a *table-locator-reference* is specified, the host variable represents the intermediate result table. The intermediate result table has the same structure as the table identified in *table-name*.
- If a *collection-derived-table* is specified, the intermediate result table is a set of rows from one or more array values. For more information, see "collection-derived-table" on page 787.
- If an *xmltable-expression* is specified, the intermediate result table is the set of rows that are returned by the "XMLTABLE" on page 755 function.
- If a *joined-table* is specified, the intermediate result table is the result of one or more join operations. For more information, see "joined-table" on page 791.

Each *table-name* or *view-name* specified in every **FROM** clause of the same SQL statement must identify a table or view that exists at the same DB2 subsystem. If a **FROM** clause is specified in a subquery of a basic predicate, a view that includes **GROUP BY** or **HAVING** must not be identified.

A *table-reference* must not identify a table that was implicitly created for an XML column.

table-locator-reference
: Each *table-locator-variable* must specify a host variable with a table locator type. The only way to assign a value to a table locator is to pass the old or new transition table of a trigger to a user-defined function or stored procedure. A table locator host variable must not have a null indicator and must not be a parameter marker. In addition, a table locator can be used only in a manipulative SQL statement.

nested-table-expression
: A *fullselect* in parentheses is called a *nested table expression*. If a nested table expression is specified, the result table is the result of that *nested-table-expression*. The columns of the result do not need unique names, but

a column with a non-unique name cannot be referenced. At any time, the table consists of the rows that would result if the fullselect were executed.

table-function-reference
If a *function-name* is specified, the result table is the set of rows returned by the table function.

expression must not contain a scalar fullselect, a function, or a reference to a column.

Each *function-name*, together with the types of its arguments, must resolve to a table function that exists at the same DB2 subsystem. An algorithm called function resolution, which is described in "Function resolution" on page 234, uses the function name and the arguments to determine the exact function to use. Unless given column names in the *correlation-clause*, the column names for a table function are those specified on the RETURNS clause of the CREATE FUNCTION statement. This is analogous to the column names of a table, which are defined in the CREATE TABLE statement.

If a column mask is used to mask the column values in the final result table, and if the result of the table function is used to derive the final result table, the column mask cannot be applied to a column that is specified in the argument of the table function.

table-UDF-cardinality-clause
The *table-UDF-cardinality clause* can be specified to each user-defined table function reference within the table spec of the FROM clause in a subselect. This option indicates the expected number of rows to be returned only for the SELECT statement that contains it.

CARDINALITY *integer-constant* specifies an estimate of the expected number of rows returned by the reference to the user-defined function. The value of *integer-constant* must range from 0 to 2147483647.

The value set in the CARDINALITY field of SYSIBM.SYSROUTINES for the table function name is used as the reference cardinality value. The product of the specified **CARDINALITY MULTIPLIER** *numeric-constant* and the reference cardinality value are used by DB2 as the expected number of rows returned by the table function reference.

In this case, the *numeric-constant* can be in the integer, decimal, or floating-point format. The value must be greater than or equal to zero. If the decimal number notation is used, the number of digits can be up to 31. An integer value is treated as a decimal number with no fraction. The maximum value allowed for a floating-point number is about 7.237E + 75. If no value has been set in the CARDINALITY field of SYSIBM.SYSROUTINES, its default value is used as the reference cardinality value. If zero is specified or the computed cardinality is less than 1, DB2 assumes that the cardinality of the reference to the user-defined table function is 1.

Only a numeric constant can follow the keyword **CARDINALITY** or **CARDINALITY MULTIPLIER**. No host variable or parameter marker is allowed in a cardinality option. Specifying a cardinality option in a table function reference does not change the corresponding CARDINALITY field in SYSIBM.SYSROUTINES. The CARDINALITY field value in SYSIBM.SYSROUTINES can be initialized by the **CARDINALITY** option in the CREATE FUNCTION (external table) statement when a user-defined

table function is created. It can be changed by the **CARDINALITY** option in the ALTER FUNCTION statement or by a direct update operation to SYSIBM.SYSROUTINES.

data-change-table-reference
A *data-change-table-reference* clause specifies an intermediate result table. This table is based on the rows that are directly changed by the SQL data change statement that is included in the clause. A *data-change-table-reference* can only be specified as the only *table-reference* in the FROM clause of the outer fullselect that is used in a select-statement and that fullselect must be in a subselect, or a SELECT INTO statement. A *data-change-table-reference* in a SELECT statement of a cursor makes the cursor read only. The target table or view of the SQL data change statement is a table or view that is reference in the query. The privileges that are held by the authorization ID of the statement must include the SELECT privilege on that target table or view.

If row access control is enforced for the target of the data change statement, the rows in the intermediate result table already satisfy the rules that are specified in the enabled row permissions. If column access control is enforced for the target of the data change statement, the enabled column masks are applied to the outermost select list. See *"select-clause" on page 765* for more information. If an INCLUDE clause is specified as part of the SQL data change statement, and these additional columns appear in the outermost select list, the column values must not be derived from columns for which column masks are defined.

Expressions in the select list of a view in a table reference can only be selected if OLD TABLE is specified or if the expression does not include any of the following objects:

- a function that is defined to read or modify SQL data
- a function that is defined as not deterministic or has an external action
- a NEXT VALUE expression for a sequence

FINAL TABLE
Specifies that the rows of the intermediate result table represent the set of rows that are changed by the SQL data change statement as they appear at the completion of the SQL data change statement. If there are AFTER triggers that result in further operations on the table that is the target of the SQL data change statement, an error is returned. If the target of the SQL data change statement is a view that is defined with an INSTEAD OF trigger for the type of data change, an error is returned.

OLD TABLE
The rows of the intermediate result table represent the set of affected rows as they exist prior to the application of the SQL data change statement.

INSERT statement
Specifies an INSERT statement as described in "INSERT" on page 1734. A fullselect in the INSERT statement cannot contain correlated references to columns that are outside of the fullselect of the INSERT statement. The target of the INSERT statement must be a base table, a view that is defined with the **WITH CASCADED CHECK** clause, or a view where the view definition has no **WHERE** clause. If there are input variables elsewhere in the fullselect, the INSERT statement cannot be a multiple row not atomic insert, or a multiple row atomic insert that specifies the **USING DESCRIPTOR** clause.

MERGE statement
Specifies a MERGE statement as described in "MERGE" on page 1760. A

table reference in the MERGE statement must not contain correlated references to columns that are outside of the table reference in the MERGE statement.

If the MERGE statement is used in the SELECT statement and the MERGE statement references a view, the view must be defined using the **WITH CASCADED CHECK OPTION** clause.

The target table or view of the MERGE statement must not have a column with a ROWID, LOB, or XML data type.

AFTER triggers that result in further operations on the target table cannot exist on the target table.

searched UPDATE statement
Specifies a searched UPDATE statement as described in "UPDATE" on page 1933. A WHERE clause or a SET clause in the UPDATE statement cannot contain correlated referenced to columns that are outside of the UPDATE statement. The target of the UPDATE statement must be a base table, a symmetric view, or a view where the view definition has no WHERE clause.

If the searched UPDATE statement is used in the SELECT statement and the UPDATE statement references a view, the view must be defined using the WITH CASCADED CHECK OPTION clause.

A searched UPDATE statement in a SELECT statement will not clear the AREO* status of a table.

AFTER triggers that result in further operations on the target table cannot exist on the target table.

searched DELETE statement
Specifies a searched DELETE statement as described in "DELETE" on page 1573. A WHERE clause in the DELETE statement cannot contain correlated references to columns that are outside of the DELETE statement. The target of the DELETE statement must be a base table, a symmetric view, or a view where the view definition has no WHERE clause.

If the searched DELETE statement is used in the SELECT statement and the DELETE statement references a view, the view must be defined using the WITH CASCADED CHECK OPTION clause.

AFTER triggers that result in further operations on the target table cannot exist on the target table.

The content of the intermediate result table for a table reference that contains an SQL data change statement is determined when the cursor is opened. The intermediate result table includes a column for each of the columns of the target table (including implicitly hidden columns) or view. All of the columns of the target table or view of an SQL data change statement are accessible by using the names of the columns from the target table or view unless the columns are renamed by using the correlation clause. If a *correlation-name* is not specified, the column names can by qualified by the target table or view name of the SQL data change statement, without the schema qualifier. If an INCLUDE clause is specified as part of the SQL data change statement, the intermediate result table will contain these additional columns.

correlation-clause
Each *correlation-name* in a *correlation-clause* defines a designator for the

immediately preceding result table, which can be used to qualify references to the columns of the table. See "correlation-clause" on page 784 for more information.

typed-correlation-clause
A *typed-correlation-clause* defines the appearance and contents of the table generated by a generic table function. This clause must be specified when the *table-function-reference* is a generic table function and cannot be specified for any other table reference. See "typed-correlation-clause" on page 786 for more information.

xmltable-expression
Specifies an invocation of the built-in XMLTABLE function. See "XMLTABLE" on page 755 for more information.

If a column mask is used to mask the column values in the final result table, and if the result of the XMLTABLE function is used to derive the final result table, the column mask cannot be applied to a column that is specified in the PASSING clause of the XMLTABLE function.

collection-derived-table
A *collection-derived-table* is used to convert the elements of one or more arrays into column values in separate rows of an intermediate result table, as explained in "collection-derived-table" on page 787.

joined-table
If a *joined-table* is specified, the result table is the result of one or more join operations as explained in "joined-table" on page 791.

period-specification
Specifies that a period specification applies to the *table-reference*. The same period name (SYSTEM_TIME or BUSINESS_TIME) must not be specified more than one time for the same table. If the table reference specifies a view, the definition of that view must not reference a user-defined function.

The rows of the table reference are derived by application of the specified period specification.

The rows of a view reference are derived by application of the specified period specifications to all of the temporal tables that are accessed when computing the result table of the view. If the view does not access any temporal tables, the period specification has no effect on the result table of the view.

If the table is a bitemporal table and a *period-specification* is not specified for both SYSTEM_TIME or BUSINESS_TIME, the table reference includes all current rows of the table and does not include any historical rows of the table.

If the CURRENT TEMPORAL SYSTEM_TIME special register is set to a value other than the null value, a *period-specification* for a table or view cannot reference SYSTEM_TIME. This restriction applies even if the view body does not reference a system-period temporal table. The exception is if the value in effect for the SYSTIMSENSITIVE bind option is NO. In this case, the *period-specification* can reference SYSTEM_TIME.

If the CURRENT TEMPORAL BUSINESS_TIME special register is set to a value other than the null value, a *period-specification* for a table or view cannot reference BUSINESS_TIME. This restriction applies even if the view body does not reference an application-period temporal table. The exception is if the value in effect for the BUSTIMESENSITIVE bind option is NO. In this case, the *period-specification* can reference BUSINESS_TIME.

Related information:

"CURRENT TEMPORAL BUSINESS_TIME" on page 194

"CURRENT TEMPORAL SYSTEM_TIME" on page 196

FOR SYSTEM_TIME

Specifies that the SYSTEM_TIME period is used for the *period-specification*. The table reference must be a system-period temporal table or a view.

Do not specify FOR SYSTEM_TIME if the value of the CURRENT TEMPORAL SYSTEM_TIME special register is not NULL.

FOR BUSINESS_TIME

Specifies that the BUSINESS_TIME period is used for the *period-specification*. The table reference must be an application-period temporal table or a view.

Do not specify FOR BUSINESS_TIME if the value of the CURRENT TEMPORAL BUSINESS_TIME special register is not NULL.

AS OF *value*

Specifies that the *table-reference* includes each row for which the begin value for the specified period is less than or equal to *value* and the end value for the period is greater than *value*.

value

Specifies an expression that returns a value of a built-in data type. The result of the expression must be comparable to the data type of the columns of the specified period according to the comparison rules specified in "Assignment and comparison" on page 121.

The expression must not have a timestamp precision that is greater than the precision of the columns for the period.

If the begin and end columns of the period are defined as TIMESTAMP WITHOUT TIME ZONE, the expression must not return a value of a timestamp with a time zone.The expression can contain any of the following supported operands:

- A constant
- A special register
- A variable (host variable, SQL variable, SQL parameter, or transition variable)
- An array element specification
- A built-in scalar function whose arguments are supported operands
- A CAST specification where the cast operand is a supported operand
- An expression that uses arithmetic operators and operands

A period specification for a view must not contain a global variable or an untyped parameter marker.

FROM *value1* **TO** *value2*

Specifies that the *table-reference* includes rows that exist for the period that is specified from *value1* up to *value2*. A row is included in the *table-reference* if the start value for the period in the row is less than *value2* and the end value for the period in the row is greater than *value1*.

value1 **or** *value2*

Specifies an expression that returns a value of a built-in data type. The result of the expression must be comparable to the data type of the

columns of the specified period according to the comparison rules specified in "Assignment and comparison" on page 121.

The expression must not have a timestamp precision that is greater than the precision of the columns for the period.

If the begin and end columns of the period are defined as TIMESTAMP WITHOUT TIME ZONE, the expression must not return a value of a timestamp with a time zone.The expression can contain any of the following supported operands:

- A constant
- A special register
- A variable (host variable, SQL variable, SQL parameter, or transition variable)
- An array element specification
- A built-in scalar function whose arguments are supported operands
- A CAST specification where the cast operand is a supported operand
- An expression that uses arithmetic operators and operands

A period specification for a view must not contain a global variable or an untyped parameter marker.

BETWEEN *value1* **AND** *value2*

Specifies that the *table-reference* includes rows in which the specified period overlaps at any point in time between *value1* and *value2*. A row is included in the *table-reference* if the start value for the period in the row is less than or equal to *value2* and the end value for the period in the row is greater than *value1*. The table reference contains zero rows if *value1* is greater than *value2*. If *value1* = *value2*, the expression is equivalent to AS OF *value1*. If *value1* or *value2* is the null value, the table reference is an empty table.

value1 **or** *value2*

Specifies an expression that returns a value of a built-in data type. The result of the expression must be comparable to the data type of the columns of the specified period according to the comparison rules specified in "Assignment and comparison" on page 121.

The expression must not have a timestamp precision that is greater than the precision of the columns for the period.

If the begin and end columns of the period are defined as TIMESTAMP WITHOUT TIME ZONE, the expression must not return a value of a timestamp with a time zone.The expression can contain any of the following supported operands:

- A constant
- A special register
- A variable (host variable, SQL variable, SQL parameter, or transition variable)
- An array element specification
- A built-in scalar function whose arguments are supported operands
- A CAST specification where the cast operand is a supported operand
- An expression that uses arithmetic operators and operands

A period specification for a view must not contain a global variable or an untyped parameter marker.

Notes

Correlated references in *table-references*:

In general, nested table expressions and table functions can be specified in any FROM clause. Columns from the nested table expressions and table functions can be referenced in the select list and in the rest of the fullselect using the correlation name. The scope of this correlation name is the same as correlation names for other table or view names in the FROM clause. The basic rule that applies for both these cases is that the correlated reference must be from a *table-reference* at a higher level in the hierarchy of subqueries.

Nested table expressions can be used in place of a view to avoid creating a view when general use of the view is not required. They can also be used when the result table is based on host variables.

For table functions, an additional capability exists. A table function can contain one or more correlated references to other tables in the same FROM clause if the referenced tables precede the reference in the left-to-right order of the tables in the FROM clause. The same capability exists for nested table expressions if the optional keyword TABLE is specified; otherwise, only references to higher levels in the hierarchy of subqueries is allowed.

A nested table expression or table function that contains correlated references to other tables in the same FROM clause:

- Cannot participate in a FULL OUTER JOIN or a RIGHT OUTER JOIN
- Can participate in LEFT OUTER JOIN or an INNER JOIN if the referenced tables precede the reference in the left-to-right order of the tables in the FROM clause

The following table shows some examples of valid and invalid correlated references. TABF1 and TABF2 represent table functions.

Table 90. Examples of correlated references

Subselect	Valid	Reason

`SELECT T.C1, Z.C5`
`FROM TABLE( TABF1(T.C2) ) AS Z, T`
`WHERE T.C3 = Z.C4;`
	No	`T.C2` cannot be resolved because `T` does not precede `TABF1` in FROM

`SELECT T.C1, Z.C5`
`FROM T, TABLE( TABF1(T.C2) ) AS Z`
`WHERE T.C3 = Z.C4;`
	Yes	`T` precedes `TABF1` in FROM, making `T.C2` known

`SELECT A.C1, B.C5`
`FROM TABLE( TABF2(B.C2) ) AS A,`
`TABLE( TABF1(A.C6) ) AS B`
`WHERE A.C3 = B.C4;`
	No	`B` in `B.C2` cannot be resolved because the table function that would resolve it, `TABF1`, follows its reference in `TABF2` in FROM

`SELECT D.DEPTNO, D.DEPTNAME,`
`EMPINFO.AVGSAL, EMPINFO.EMPCOUNT`
`FROM DEPT D,`
`(SELECT AVG(E.SALARY) AS AVGSAL,`
`COUNT(*) AS EMPCOUNT`
`FROM EMP E`
`WHERE E.WORKDEPT = D.DEPTNO)`
`AS EMPINFO;`
	No	`DEPT` precedes nested table expression, but keyword TABLE is not specified, making `D.DEPTNO` unknown

Table 90. Examples of correlated references (continued)

Subselect	Valid	Reason

`SELECT D.DEPTNO, D.DEPTNAME,`
`EMPINFO.AVGSAL, EMPINFO.EMPCOUNT`
`FROM DEPT D,`
`TABLE (SELECT AVG(E.SALARY) AS AVGSAL,`
`COUNT(*) AS EMPCOUNT`
`FROM EMP E`
`WHERE E.WORKDEPT = D.DEPTNO)`
`AS EMPINFO;`
	Yes	DEPT precedes nested table expression and keyword TABLE is specified, making D.DEPTNO known

Affects of special registers:

The setting of the CURRENT TEMPORAL BUSINESS_TIME and CURRENT TEMPORAL SYSTEM_TIME special registers might affect the result of a query, as described in the following situations:

- Assume the following conditions:
  - A table reference is an application-period temporal table.
  - The columns of the BUSINESS_TIME period are defined as TIMESTAMP(6).
  - The CURRENT TEMPORAL BUSINESS_TIME special register is set to a non-null value.

  In this case, a query is executed as if it contained the following specification:

  ```
  FOR BUSINESS_TIME AS OF CURRENT TEMPORAL BUSINESS_TIME
  ```
- Assume the following conditions:
  - A table reference is an application-period temporal table.
  - The columns of the BUSINESS_TIME period are defined as DATE.
  - The CURRENT TEMPORAL BUSINESS_TIME special register is set to a non-null value.

  In this case, a query is executed as if it contained the following specification:

  ```
  FOR BUSINESS_TIME AS OF CAST(CURRENT TEMPORAL BUSINESS_TIME AS DATE)
  ```
- If the CURRENT TEMPORAL SYSTEM_TIME special register is set to a non-null value, a query is executed as if it contained the following specification:

  ```
  FOR SYSTEM_TIME AS OF CURRENT TEMPORAL SYSTEM_TIME
  ```

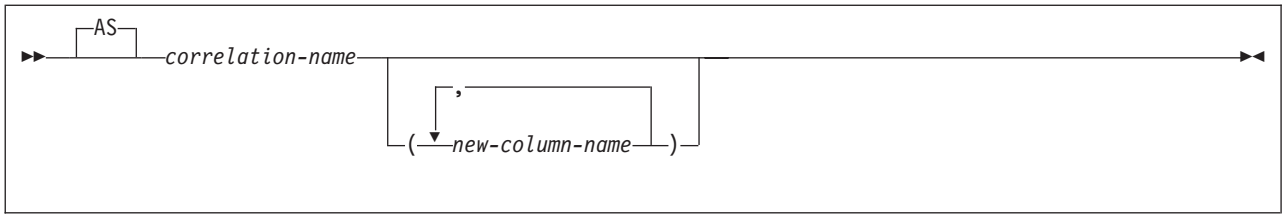
Related reference:

"Examples of subselects" on page 805

correlation-clause:

Each *correlation-name* in a *correlation-clause* defines a designator for the immediately preceding result table, which can be used to qualify references to the columns of the table.

correlation-clause:



The preceding result table is one of the following objects:

- A table
- A view
- A nested table expression
- A table function
- A data-change table reference
- A collection-derived table

new-column-name is an SQL identifier of 128 UTF-8 bytes or less. Using *new-column-name* to list and rename the columns is optional. A correlation name must be specified for nested table expressions and references to table functions.

If *correlation-name* is not specified for a data-change table reference, the correlation name is the name of the target table or view of the SQL data change statement. Otherwise, the correlation name is *correlation-name*.

If a *new-column-name* list is specified in *correlation-clause*, the number of names must be the same as the number of columns in the corresponding object. Each name must be unique and unqualified. If columns are added to an underlying table of a *table-reference*, the number of columns in the result of the *table-reference* no longer matches the number of names in its *correlation-clause*. Therefore, when a rebind of a package containing the query in question is attempted, DB2 returns an error and the rebind fails. At that point, change the *correlation-clause* of the embedded SQL statement in the application program so that the number of names matches the number of columns. Then prepare the modified program again.

An exposed name is a *correlation-name* or a *table-name* or view name that is not followed by a *correlation-name*. The exposed names in a FROM clause must be unique. Any qualified reference to a column for a table, view, nested table expression, table function, data-change table reference, or collection-derived table must use the exposed name.

If the same table name or view name is specified twice, at least one specification must be followed by a *correlation-name*. The *correlation-name* is used to qualify references to the columns of the table or view.

When a *correlation-name* is specified, column names can also be specified to give names to the columns of the *table-name*, *view-name*, *nested-table-expression*, *table-function*, *data-change-table-reference*, or *collection-derived-table*. If a column list is specified, there must be a name in the column list for each column in the table or view and for each result column in the *table-function*, *data-change-table-reference*, or *collection-derived-table*.

For more information, see “Correlation names” on page 209.

In general, *nested-table-expression*, *table-function*, *data-change-table-reference*, or *collection-derived-table* can be specified in any FROM clause. Columns from the *nested-table-expression*, *table-function*, *data-change-table-reference*, or *collection-derived-table* can be referenced in the SELECT list and in the rest of the subselect using a correlation name. The scope of this correlation name is the same as correlation names for other table or view names in the FROM clause.

Related reference:

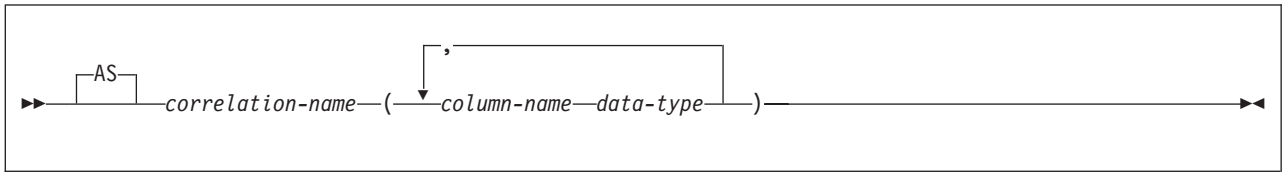
"SET assignment-statement" on page 1875

"table-reference" on page 773

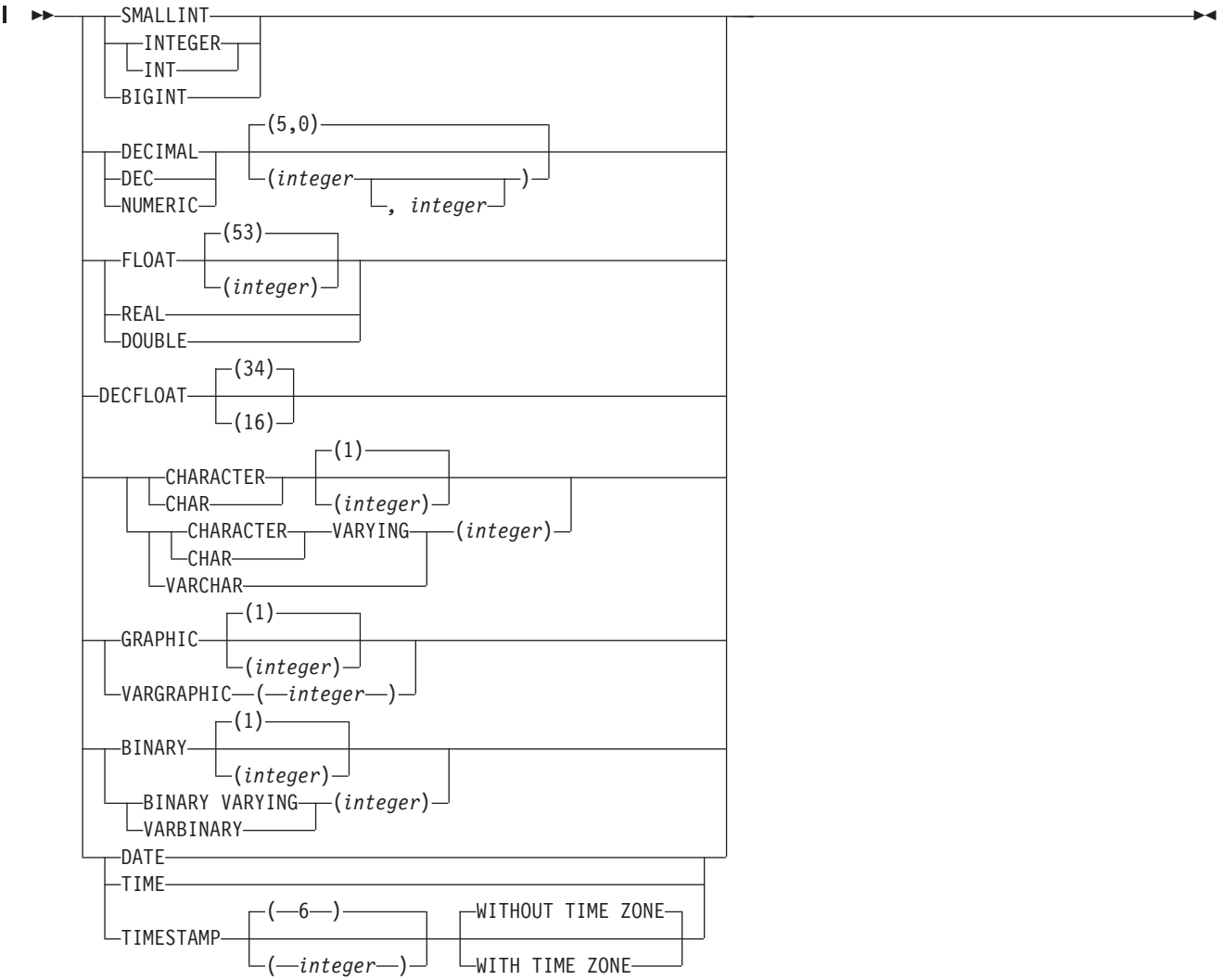
typed-correlation-clause:

A *typed-correlation-clause* defines the appearance and contents of the table generated by a generic table function.

typed-correlation-clause:



data-type:



typed-correlation-clause

The *typed-correlation-clause* defines the appearance and contents of the table that is generated by a generic table function. *typed-correlation-clause* must be specified when the *table-function-reference* is a generic table function and cannot be specified for any other table reference.

The maximum number of columns specified in the *typed-correlation-clause* is 750, an error is returned if the number of specified columns exceeds the limit.

An error is returned if duplicate column names are specified in a *typed-correlation-clause*.

An error is returned if the data type that is specified for a column name is not one of the supported data types for a generic table function.

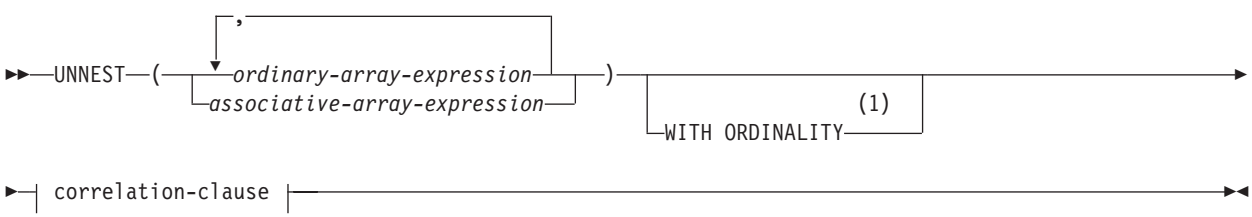
Related reference:

"CREATE FUNCTION (external table)" on page 1191

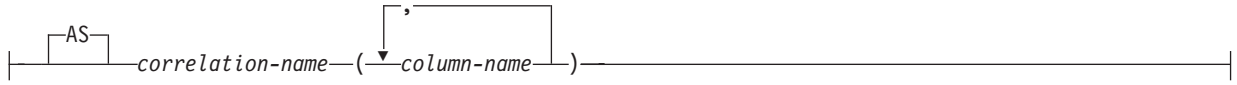
collection-derived-table

A *collection-derived table* is used to convert the elements of one or more arrays into column values in separate rows of an intermediate result table.

collection-derived-table:



correlation-clause:



Notes:

1 WITH ORDINALITY can be specified only if the argument to UNNEST is *ordinary-array-expression*. *associative-array-expression* must not be specified when WITH ORDINALITY is also specified.

WITH ORDINALITY
Specifies that an extra column of data type INTEGER is returned as the last column in the result table. This column contains the position of the element in the array.

correlation-clause
Specifies the correlation name that is to be used as a table designator for the result table of the collection derived table, and a list of column names for the result table. The correlation name can be used to qualify references to the columns of the result table.

The result columns can be referenced in the SELECT list, and in the rest of the subselect by using the names that are specified for the columns in the correlation clause.

A collection-derived table can be specified as a table reference in a FROM clause, in a context where arrays are supported.

The intermediate result table is derived as follows:

- If a single expression that returns an ordinary array is specified, the intermediate result table is a single-column table with a column data type that matches the array element data type.
- If multiple expressions that return an ordinary array are specified:
  - The first array provides the first column in the result table, the second array provides the second column, and so on.
  - The data type of each column matches the data type of the array elements of the corresponding array argument.
  - If the cardinalities of the arrays are not identical, the cardinality of the resulting table is the same as the array with the largest cardinality.

- The column values in the table are set to the null value for all rows whose array index value is greater than the cardinality of the corresponding array. In other words, if each array is viewed as a table with two columns, one for the array indexes and one for the data, UNNEST performs an outer join among the arrays, using equality on the array indexes as a join predicate.

- If a single *associative-array-expression* or an *array-function-invocation* that returns an associative array is specified:
  - The intermediate result table is a table with two columns, where the first column data type matches the array index data type, and the second column data type matches the array element data type.
  - The first column contains the indexes of the elements in the array.
  - The second column contains the elements in the array.
  - The columns can be referenced in the SELECT list and the in rest of the subselect by using the names that are specified for the columns in the *correlation-clause*.
- If all arguments are null arrays, the result is an empty table.

The intermediate result table that is produced by an invocation of UNNEST must not result in more than 750 columns.

An *array-function-invocation* is a function invocation that resolves to a function that returns an ordinary or an associative array type. An *array-function-invocation* must not include a reference to a column of a common table expression.

ordinary-array-expression
Specifies one of the following items:
- An SQL variable
- An SQL parameter
- An *array-function-invocation*
- A CAST specification of a parameter marker to an ordinary array type

associative-array-expression
Specifies one of the following items:
- An SQL variable
- An SQL parameter
- An *array-function-invocation*
- A CAST specification of a parameter marker to an associative array type

Names for the result columns that are produced by an UNNEST specification can be provided as part of the *correlation-clause* of the *collection-derived-table* clause.

Example 1: Suppose that PHONENUMBERS is a user-defined array type that is defined as an ordinary array. RECENT_CALLS is an array variable of the PHONENUMBERS type. RECENT_CALLS contains the following phone numbers:
- 9055553907
- 4165554213
- 4085553678

The following SELECT statement uses UNNEST to retrieve the list of phone numbers from the array:

```
SELECT T.ID, T.NUM
 FROM UNNEST(RECENT_CALLS) WITH ORDINALITY AS T(NUM, ID);
```

The WITH ORDINALITY clause indicates that the result table is to include an additional column that reflects the ordinal position of each array element in the array. The additional column is the last column of the result table from the UNNEST operation. The correlation clause that follows the WITH ORDINALITY clause specifies that the additional column is named ID, and the array element column is named NUM. These column names can be explicitly referenced in the SELECT list of the query. The SELECT list in this example reorders the columns from the result of UNNEST. The result table looks like this:

ID	NUM
1	9055553907
2	4165554213
3	4085553678

In the SELECT statement, the columns that result from the UNNEST operation have been reordered in the SELECT list, so that the column that reflects the position of each array element is the first column of the final result table.

Example 2: Suppose that PERSONAL_PHONENUMBERS is a user-defined array type that is defined as an associative array. PHONELIST is an array variable of the PERSONAL_PHONENUMBERS type. Values have been assigned to the elements of PHONELIST with the following statements:

```
SET PHONELIST['Home'] = '4443051234';
SET PHONELIST['Work'] = '4443052345';
SET PHONELIST['Cell'] = '4447893456';
```

The following SELECT statement is executed:

```
SELECT T.ID, T.PHONE
 FROM UNNEST(PHONELIST) AS T(ID, PHONE);
```

The result table looks like this, although the order of rows might differ:

ID	PHONE
Cell	4447893456
Home	4443051234
Work	4443052345

Example 3: Suppose that PHONES and IDS are two SQL variables with array values of the same cardinality. The following SQL statement converts the array contents into a table with three columns (one for each array and one for the position), and one row for each array element.

The following SELECT statement is executed:

```
SELECT T.PHONE, T.ID, T.INDEX FROM UNNEST(PHONES, IDS)
WITH ORDINALITY AS T(PHONE, ID, INDEX)
ORDER BY T.INDEX;
```

Related reference:

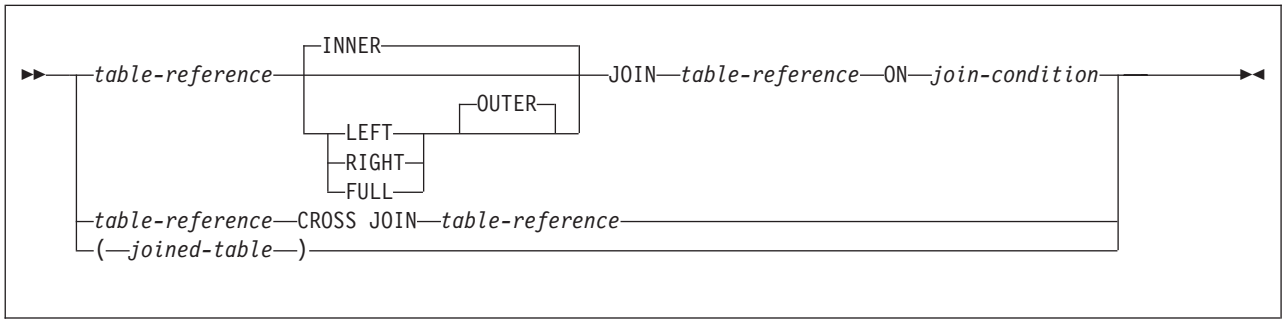
"SET assignment-statement" on page 1875

"table-reference" on page 773

joined-table

A *joined-table* specifies an intermediate result table that is the result of either an inner join, an outer join, or a cross join. The table is derived by applying one of the join operators: INNER, LEFT OUTER, RIGHT OUTER, FULL OUTER, or CROSS to its operands.

joined-table



Cross joins represent the cross product of the tables, where each row of the left table is combined with every row of the right table. Inner joins can be thought of as the cross product of the tables, keeping only the rows where the join condition is true. The result table might be missing rows from either or both of the joined tables. Outer joins include the rows produced by the inner join as well as the missing rows, depending on the type of outer join as follows:

Left outer join
Includes rows from the left table that were missing from the inner join.

Right outer join
Includes rows from the right table that were missing from the inner join.

Full outer join
Includes rows from both the left and right tables that were missing from the inner join.

If a join operator is not specified, INNER is the default. The order in which a LEFT OUTER JOIN or RIGHT OUTER JOIN is performed can affect the result.

If FULL OUTER JOIN is specified, a Unicode column in an EBCDIC table must not be referenced in join-condition.

A *joined-table* can be used in any context in which any form of the SELECT statement is used. Both a view and a cursor is read-only if its SELECT statement includes a *joined-table*.

If LEFT OUTER JOIN, RIGHT OUTER JOIN, or FULL OUTER JOIN is specified:

- A ROW CHANGE TIMESTAMP expression can only be referenced in a subselect of the outer join if the table designator identifies a base table that includes a row change timestamp column.

- The RID built-in function and the ROW CHANGE TOKEN expression must not be specified in the subselect that contains the FROM clause.

Related reference:

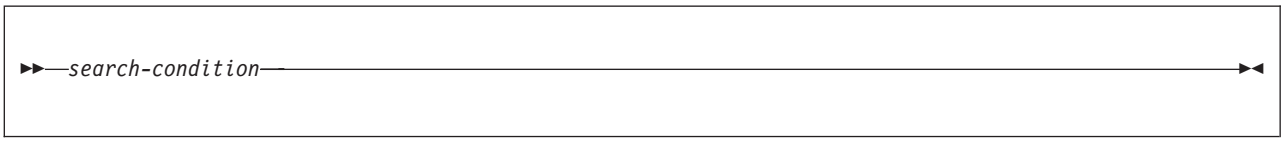
"Examples of subselects" on page 805

join-condition:

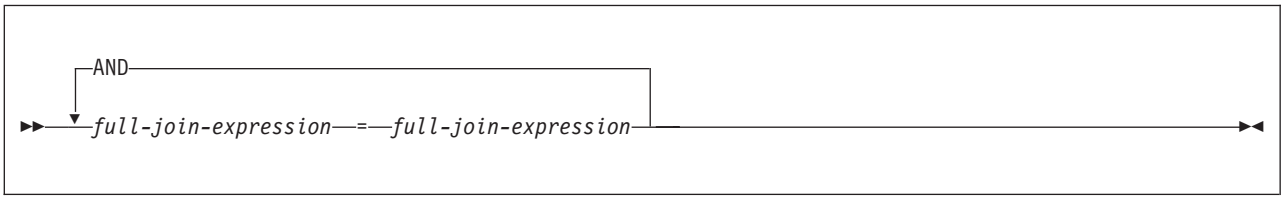
join-condition specifies the conditions of a join that is used in a query.

join-condition

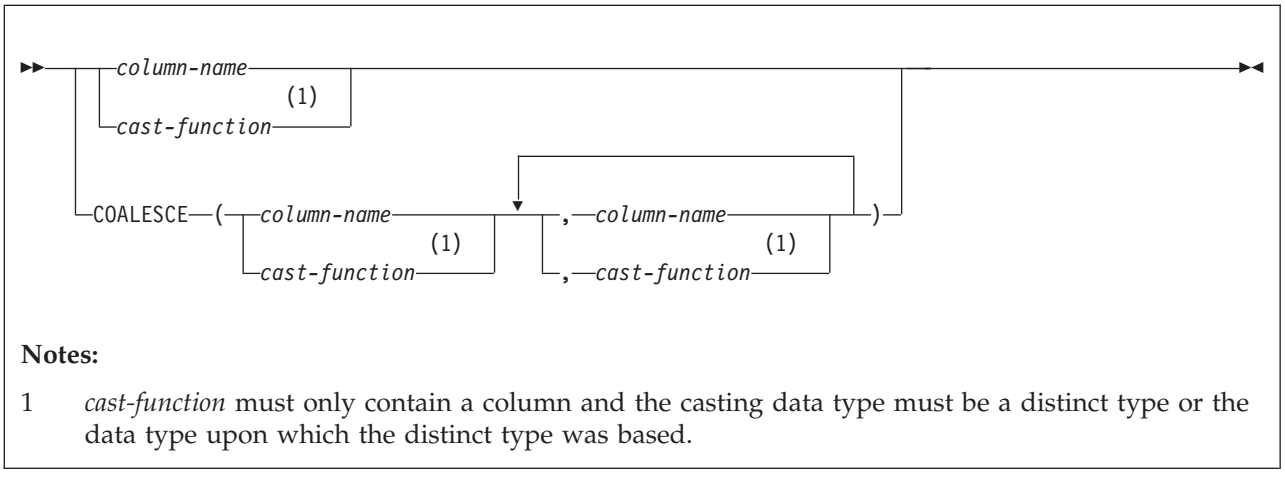
For INNER, LEFT OUTER, and RIGHT OUTER joins:



For FULL OUTER joins:



full-join-expression:



Notes:

1 *cast-function* must only contain a column and the casting data type must be a distinct type or the data type upon which the distinct type was based.

For INNER, LEFT OUTER, and RIGHT OUTER joins, the *join-condition* is a *search-condition* that must conform to these rules:

- With one exception, It cannot contain any subqueries. If the *join-table* that contains the *join-condition* in the associated FROM clause is composed of only INNER joins, the *join-condition* can contain subqueries.
- Any column that is referenced in an expression of the *join-condition* must be a column of one of the operand tables of the associated join operator (in the scope of the same *joined-table* clause).

For a FULL OUTER (or FULL) join, the *join-condition* is a search condition in which the predicates can only be combined with AND. In addition, each predicate must have the form 'expression = expression', where one expression references only columns of one of the operand tables of the associated join operator, and the other expression references only columns of the other operand table. The values of the expressions must be comparable.

Each *full-join-expression* in a FULL OUTER join must include a column name or a cast function that references a column. The COALESCE function is allowed.

For any type of join, column references in an expression of the *join-condition* are resolved using the rules for resolution of column name qualifiers specified in "Resolution of column name qualifiers and column names" on page 212 before any rules about which tables the columns must belong to are applied.

Related reference:

"Examples of subselects" on page 805

Join operations:

A *join-condition* specifies pairings of T1 and T2, where T1 and T2 are the left and right operand tables of its associated JOIN operator. For all possible combinations of rows T1 and T2, a row of T1 is paired with a row of T2 if the *join-condition* is true.

When a row of T1 is joined with a row of T2, a row in the result consists of the values of that row of T1 concatenated with the values of that row of T2. The execution might involve the generation of a null row. The *null row* of a table consists of a null value for each column of the table, regardless of whether the columns allow null values.

The following summarizes the results of the join operations:

- The result of T1 INNER JOIN T2 consists of their paired rows.
- The result of T1 LEFT OUTER JOIN T2 consists of their paired rows and, for each unpaired row of T1, the concatenation of that row with the null row of T2. All columns derived from T2 allow null values.
- The result of T1 RIGHT OUTER JOIN T2 consists of their paired rows and, for each unpaired row of T2, the concatenation of that row with the null row of T1. All columns derived from T1 allow null values.
- The result of T1 FULL OUTER JOIN T2 consists of their paired rows and, for each unpaired row of T1, the concatenation of that row with the null row of T2, and for each unpaired row of T2, the concatenation of that row with the null row in T1. All columns of the result table allow null values.
- The result of T1 CROSS JOIN T2 consists of each row of T1 paired with each row of T2. CROSS JOIN is also known as Cartesian product.

A join operation is part of a FROM clause. For the purpose of predicting which rows will be returned from a SELECT statement containing a join operation, assume that the join operation is performed before the other clauses in the statement.

A cross join can also be specified without the CROSS JOIN syntax, by listing the two tables in the FROM clause separated by commas without using a WHERE clause to supply join criteria.

Related reference:

"Examples of subselects" on page 805

where-clause

The WHERE clause specifies a result table that consists of those rows of R for which the search condition is true. R is the result of the FROM clause of the subselect.

where-clause

The search condition must conform to the following rules:

- Each column name must unambiguously identify a column of R or be a correlated reference. A column name is a correlated reference if it identifies a column of a table, view, *common-table-expression*, or *nested-table-expression* that is identified in an outer subselect.
- An aggregate function must not be specified unless the WHERE clause is specified in a subquery of a HAVING clause and the argument of the function is a correlated reference to a group.

Any subquery in the *search-condition* is effectively executed for each row of R and the results are used in the application of the *search-condition* to the given row of R. A subquery is actually executed for each row of R only if it includes a correlated reference. In fact, a subquery with no correlated references is executed just one time, whereas a subquery with a correlated reference might have to be executed one time for each row.

If row access controls are enabled for a table and no other row permission is defined, the row access control search condition is the default row permission, 1 = 0. If only one row permission is defined, the row access control search condition is the search conditions that are specified by that permission. Otherwise, if multiple row permissions are defined for a table, the row access control search condition is derived by application of the logical OR operator to the search conditions that are specified by each row permission. This row access control search condition, as a whole, is connected by application of the logical AND operator to the search conditions specified by the WHERE clause and has the same precedence level as other search conditions in the WHERE clause. This process is repeated for each *table-reference* in the FROM clause of the subselect for which row access controls are enabled.

The row access control search condition acts as a filter to the *table-reference* to determine the results of the *table-reference* that are accessible to the authorization ID or role of the subselect. Because the order in which operators are evaluated is undefined for operators at the same precedence level, other search conditions in the WHERE clause might be evaluated before the row access control search condition. So, the other search conditions have access to the rows that are restricted by the row permission rules. To ensure that sensitive data is protected, the predicates that reference user-defined functions that are defined with the NOT SECURED option are always evaluated after the row access control search condition.

The column access control does not affect the operation of the WHERE clause.

Related reference:
"Examples of subselects" on page 805

group-by-clause

The GROUP BY clause specifies a result table that consists of a grouping of the rows of intermediate result table that is the result of the previous clause.

group-by-clause

In its simplest form, a GROUP BY clause contains a *grouping-expression*.

A *grouping-expression* is an expression that defines the grouping of R. The following restrictions apply to *grouping-expression*:

- If *grouping-expression* is a single column, the column name must unambiguously identify a column of R.
- The result of *grouping-expression* cannot be a LOB data type (or a distinct type that is based on a LOB) or an XML data type.
- *grouping-expression* cannot include any of the following items:
  - A correlated column
  - A host variable
  - An aggregate function
  - Any function or expression that is not deterministic or that is defined to have an external action
  - A scalar fullselect
  - A CASE expression whose *searched-when-clause* contains a quantified predicate, an IN predicate using a fullselect, or an EXISTS predicate

The result of GROUP BY is a set of groups of rows. In each group of more than one row, all values of each *grouping-expression* are equal, and all rows with the same set of values of the *grouping-expression* are in the same group. For grouping, all null values for a *grouping-expression* are considered equal.

If a *grouping-expression* contains DECFLOAT values, the DECFLOAT values with the same value will be in the same group. But the number of digits returned for each group is unpredictable.

Because every row of a group contains the same value of any *grouping-expression*, a *grouping-expression* can be used in a search condition in a HAVING clause or an expression in a SELECT clause, or in a *sort-key-expression* of an ORDER BY clause. In each case, the reference specifies only one value for each group. For example, if *grouping-expression* is col1+col2, col1+col2+3 would be an allowed expression in the select list. Associative rules for expressions do not allow the similar expression of 3+col1+col2, unless parentheses are used to ensure that the corresponding expression is evaluated in the same order. Thus, 3+(col1+col2) would also be allowed in the select list. If the concatenation operator is used, *grouping-expression* must be used exactly as the expression was specified in the select list.

If a *grouping-expression* contains varying-length strings with trailing blanks, the values in the group can differ in the number of trailing blanks and might not all have the same length. In that case, a reference to *grouping-expression* still specifies only one value for each group, but the value for a group is chosen arbitrarily from the available set of values. Thus, the actual length of the result value is unpredictable.

Row access controls do not affect the operation of the GROUP BY clause.

In certain contexts, the semantics of the column mask can conflict with those in the GROUP BY clause. When this occurs, the column mask cannot be applied for the statement and an error will be returned at bind time. See the select-clause for more information about how column access controls affect the GROUP BY clause.

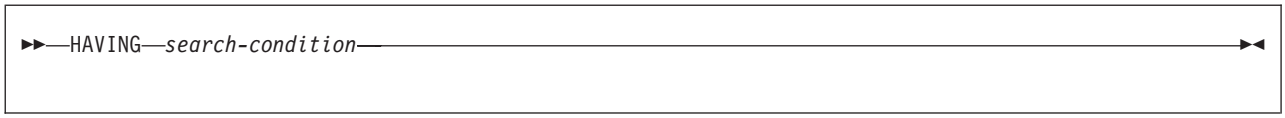
Related reference:

"Examples of subselects" on page 805

having-clause

The HAVING clause specifies a result table that consists of those groups of the intermediate result table for which the search-condition is true. The intermediate result table is the result of the previous clause. If this clause is not GROUP BY, the intermediate result table is considered a single group with no grouping columns of the previous clause of the subselect.

having-clause



Each *column-name* in *search-condition* must be one of the following:

- Unambiguously identify a grouping column of the intermediate result table
- Be specified within an aggregate function[22]
- Be a correlated reference. A *column-name* is a correlated reference if it identifies a column of a table, view, *common-table-expression*, or *nested-table-expression* that is identified in an outer subselect

A group of the intermediate result table to which the search condition is applied supplies the argument for each function in the search condition, except for any function whose argument is a correlated reference.

If the search condition contains a subquery, the subquery can be thought of as being executed each time the search condition is applied to a group of the intermediate result table, and the results used in applying the search condition. In actuality, the subquery is executed for each group only if it contains a correlated reference. For an illustration of the difference, see Example 4 and Example 5.

A correlated reference to a group of the intermediate result table must either identify a grouping column or be contained within an aggregate function.

When HAVING is used without GROUP BY, any expression or column name in the select list must appear within an aggregate function.

The RID built-in function and the ROW CHANGE expression cannot be specified in a HAVING clause unless they are within an aggregate function.

Row access controls do not affect the operation of the HAVING clause.

In certain contexts, the semantics of the column mask can conflict with those in the HAVING clause. When this occurs, the column mask cannot be applied for the statement and an error will be returned at bind time. See the select-clause for more information about how column access controls affect the HAVING clause.

Related reference:

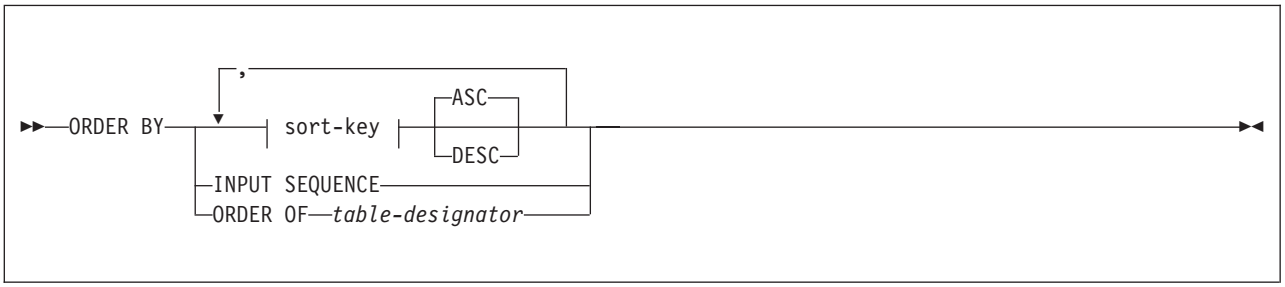
"Examples of subselects" on page 805

22. See Chapter 3, "Functions," on page 337 for restrictions that apply to the use of aggregate functions.

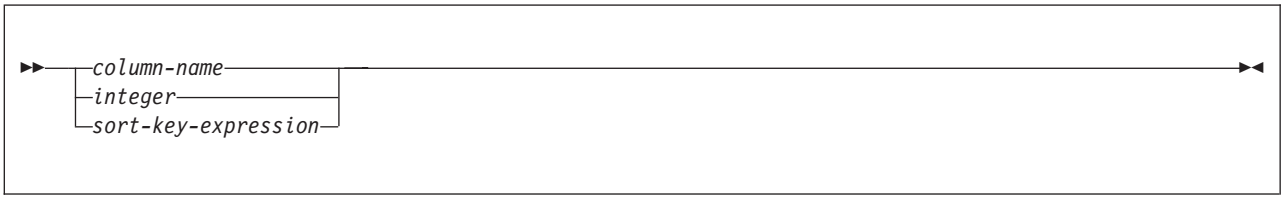
order-by-clause

The ORDER BY clause specifies an ordering of the rows of the result table.

order-by-clause



sort-key:



A subselect that contains an ORDER BY clause cannot be specified in the outermost fullselect of a view

If the subselect is not enclosed within parentheses and is not the outermost fullselect, the **ORDER BY** clause cannot be specified. The **ORDER BY** clause cannot be used in an outermost fullselect that contains a **FOR UPDATE** clause.

An **ORDER BY** clause that is specified in a subselect only effects the order of the rows that returned by the query if the subselect is the outermost fullselect, except when a nested subselect includes an **ORDER BY** clause and the outermost fullselect specifies that the ordering of the rows should be retained (by using the **ORDER OF** *table-designator* clause).

Multiple **ORDER BY** clauses can be specified in the same subselect if each clause is separated with parentheses.

INPUT SEQUENCE
Indicates that the result table reflects the input order of the rows specified in the VALUES clause of an INSERT statement. INPUT SEQUENCE ordering can be specified only when an INSERT statement is specified in a *from-clause*.

ORDER OF *table-designator*
Specifies that the same ordering of the rows for the result table that is designated by *table-designator* should be applied to the result table of the subselect (or fullselect) that contains the ORDER OF specification. There must be a table reference in the FROM clause of the subselect (or fullselect) that specifies this clause and matches *table-designator*.

sort-key

A *column-name*, *integer*, or *sort-key-expression* that specifies the value that is to be used to order the rows of the result of the subselect.

If a single *sort-key* is identified, the rows are ordered by the values of that *sort-key*. If more than one *sort-key* is identified, the rows are ordered by the values of the first *sort-key*, then by the values of the second *sort-key*, and so on. A *sort-key* cannot be a LOB or XML expression.

The result table can be ordered by a named column in the select list by specifying a *sort-key* that is an integer or the column name. The result table can be ordered by an unnamed column in the select list by specifying a *sort-key* that is an integer or, in some cases, by a *sort-key-expression* that matches the expression in the select list.

column-name

An identifier that usually identifies a column of the result table. In this case, *column-name* must be the name of a named column in the select list. If the *fullselect* includes a set operator, the column name cannot be qualified.

If the query is a *subselect*, the *column-name* can also identify a column name of a table, view, or nested table expression identified in the FROM clause, including a column that is defined as implicitly hidden. The subselect must not include any of the following:

- DISTINCT in the select list
- Aggregate functions in the select list
- GROUP BY clause

integer

An unsigned integer that must be greater than 0 and not greater than the number of columns in the result table. The integer *n* identifies the *n*th column of the result table.

sort-key-expression

An expression that is not simply a *column-name* or unsigned integer constant. The query to which ordering is applied must be a *subselect* to use this form of the *sort-key*.

The *sort-key-expression* cannot include an expression that is not deterministic or a function that is defined to have an external action except for the RID built-in function and the ROW CHANGE expression. Any column name in the expression must conform to the rules described Column names in sort keys. If *sort-key-expression* includes an aggregate function, the input arguments to that function must not reference a named column in the select list that is derived from an aggregate function. An expression cannot be specified if DISTINCT is used in the select list of the *subselect*.

If the *subselect* is grouped, the *sort-key-expression* might or might not be in the select list of the *subselect*. When *sort-key-expression* is not in the select list the following rules apply:

- Each expression in the ORDER BY clause must either:
  - Use one or more grouping expressions
  - Use a column name that either unambiguously identifies a grouping column of R or is specified within a aggregate function.
- Each expression in the ORDER BY clause must not contain a scalar fullselect.

ASC
Uses the values of the *sort-key* in ascending order.

ASC is the default.

DESC
Uses the values of the *sort-key* in descending order.

Ordering is performed in accordance with the comparison rules described in Chapter 2, "Language elements," on page 53, beginning on page "Numeric comparisons" on page 134. The null value is higher than all other values. If your ordering specification does not determine a complete ordering, rows with duplicate values of the last identified *sort-key* have an arbitrary order. If you do not specify ORDER BY, the rows of the result table have an arbitrary order.

Column access controls do not effect the operation of the ORDER BY clause. The order is based on the original column values. However, after column masks are applied, the masked values in the final result table might not reflect the order of the original column values.

Column names in sort keys: A column name in a *sort-key* must conform to the following rules:

- If the column name is qualified, the query must be a *subselect*. The column name must unambiguously identify a column of a table, view, or nested table expression in the FROM clause of the subselect; its value is used to compute the value of the sort specification.
- If the column name is unqualified and the query is a *subselect*:
  - If the column name is identical to the name of more than one column of the result table, the column name must unambiguously identify a column of some table, view, or nested table expression in the FROM clause of the ordering subselect.
  - If the column name is identical is one column of the result table, its value is used to compute the value of the sort specification.
  - If the column name is not identical to a column in the result table, it must unambiguously identify a column of a table, view, or nested table expression in the FROM clause of the subselect. If the column name is identical to one column of a table, view, or nested table expression in the FROM clause of the subselect, its value is used to compute the value of the sort specification.

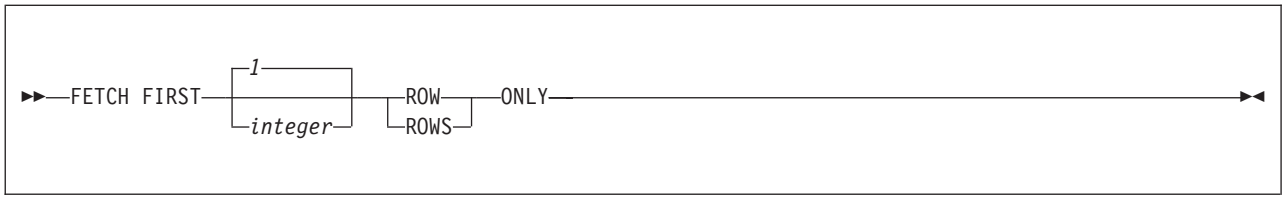
Related reference:

"Examples of subselects" on page 805

fetch-first-clause

The FETCH FIRST clause limits the number of rows that can be fetched. It improves the performance of queries with potentially large result tables when only a limited number of rows are needed.

fetch-first-clause



The FETCH FIRST clause sets a maximum number of rows that can be retrieved. FETCH FIRST specifies that only *integer* rows should be made available to be retrieved, regardless of how many rows there might be in the result table when this clause is not specified. An attempt to fetch beyond integer rows is handled the same way as normal end of data. The value of *integer* must be a positive integer (not zero). The default is 1.

The FETCH FIRST clause specifies an ordering of the rows of the result table. A subselect that contains a FETCH FIRST clause cannot be specified in the following objects:

- The outermost fullselect of a view
- The definition of a materialized query table

Limiting the result table to the first *n* rows can improve performance. The DB2 system will cease processing the query when it has determined the first *n* rows. If both the FETCH FIRST clause and the OPTIMIZE FOR clause are specified, the lower of the *integer* values from these clause will be used to influence the buffer size. The values are considered independently for optimization purposes. If the OPTIMIZE FOR clause is not specified, a default of OPTIMIZE FOR *integer* ROWS, where *integer* is the value that is specified in the FETCH FIRST clause, is assumed. The DB2 system uses this value for access path optimization.

Specification of the **FETCH FIRST** clause in an outermost fullselect makes the result table read-only. A read-only result table must not be referenced in an UPDATE, MERGE, or DELETE statement. The **FETCH FIRST** clause cannot be used in an outermost fullselect that contains a **FOR UPDATE** clause.

If the FETCH FIRST clause is specified in a subselect, and the subselect is not the outermost fullselect, the subselect must be enclosed in parentheses.

If both the FETCH FIRST clause and the ORDER BY clause are specified, the ordering is performed on the entire result table prior to returning the first *n* rows.

Multiple **FETCH FIRST** clauses can be specified in the same subselect if each clause is separated with parentheses.

If the FETCH FIRST clause is specified in the outermost fullselect of a SELECT statement that contains a data change statement (an INSERT, DELETE, UPDATE,

or MERGE statement), all rows are processed by the specified data change statement, but only the number of rows that is specified in the FETCH FIRST clause are returned in the final result table.

Row access controls can indirectly effect the FETCH FIRST clause because row access controls effect the rows that are accessible to the authorization ID or role of the subselect. Column access controls do no effect the FETCH FIRST clause.

Related concepts:

Fast implicit close (DB2 Performance)

Related tasks:

Optimizing retrieval for a small set of rows (DB2 Application programming and SQL)

Fetching a limited number of rows (DB2 Performance)

Related reference:

"Examples of subselects" on page 805

Examples of subselects

Examples of subselects can illustrate how to use the various clauses of the subselect to construct queries.

Example 1: Show all rows of the table DSN8B10.EMP.

```
SELECT * FROM DSN8B10.EMP;
```

Example 2: Show the job code, maximum salary, and minimum salary for each group of rows of DSN8B10.EMP with the same job code, but only for groups with more than one row and with a maximum salary greater than 50000.

```
SELECT JOB, MAX(SALARY), MIN(SALARY)
  FROM DSN8B10.EMP
  GROUP BY JOB
  HAVING COUNT(*) > 1 AND MAX(SALARY) > 50000;
```

Example 3: For each employee in department E11, get the following information from the table DSN8B10.EMPPROJACT: employee number, activity number, activity start date, and activity end date. Using the CHAR function, convert the start and end dates to their USA formats. Get the needed department information from the table DSN8B10.EMP.

```
SELECT EMPNO, ACTNO, CHAR(EMSTDATE,USA), CHAR(EMENDATE,USA)
  FROM DSN8B10.EMPPROJACT
  WHERE EMPNO IN (SELECT EMPNO FROM DSN8B10.EMP
                  WHERE WORKDEPT = 'E11');
```

Example 4: Show the department number and maximum departmental salary for all departments whose maximum salary is less than the average salary for all employees. (In this example, the subquery would be executed only one time.)

```
SELECT WORKDEPT, MAX(SALARY)
  FROM DSN8B10.EMP
  GROUP BY WORKDEPT
  HAVING MAX(SALARY) < (SELECT AVG(SALARY)
                        FROM DSN8B10.EMP);
```

Example 5: Show the department number and maximum departmental salary for all departments whose maximum salary is less than the average salary for employees in all other departments. (In contrast to Example 4, the subquery in this statement, containing a correlated reference, would need to be executed for each group.)

```
SELECT WORKDEPT, MAX(SALARY)
  FROM DSN8B10.EMP Q
  GROUP BY WORKDEPT
  HAVING MAX(SALARY) < (SELECT AVG(SALARY)
                        FROM DSN8B10.EMP
                        WHERE NOT WORKDEPT = Q.WORKDEPT);
```

Example 6: For each group of employees hired during the same year, show the year-of-hire and current average salary. (This example demonstrates how to use the AS clause in a FROM clause to name a derived column that you want to refer to in a GROUP BY clause.)

```
SELECT HIREYEAR, AVG(SALARY)
  FROM (SELECT YEAR(HIREDATE) AS HIREYEAR, SALARY
          FROM DSN8B10.EMP) AS NEWEMP
  GROUP BY HIREYEAR;
```

Example 7: For an example of how to group the results of a query by an expression in the SELECT clause without having to retype the expression, see Example 4 for CASE expressions.

Example 8: Get the employee number and employee name for all the employees in DSN8B10.EMP. Order the results by the date of hire.

```
SELECT EMPNO, FIRSTNME, LASTNAME
  FROM DSN8B10.EMP
  ORDER BY HIREDATE;
```

Example 9: Select all the rows from tables T1 and T2 and order the rows such that the rows from table T1 are first and are ordered by column C1, followed by the rows from T2, which are ordered by column C2. The rows of T1 are retrieved by one subselect which is connected to the results of another subselect that retrieves the rows from T2. Each subselect specifies the ordering for the rows from the referenced table. Note that both subselects need to be enclosed in parenthesis because each subselect is not the outermost fullselect.

```
(SELECT * FROM T1 ORDER BY C1)
UNION
(SELECT * FROM T2 ORDER BY C2);
```

Example 10: Specify the ORDER BY clause to order the results of a union using the second column of the result table if the union. In this example, the second ORDER BY clause applies to the results of the outermost fullselect (the result of the union) rather than to the second subselect. If the intent is to apply the second ORDER BY clause to the second subselect, the second subselect should be enclosed within parentheses as shown in Example 9.

```
(SELECT * FROM T1 ORDER BY C1)
UNION
SELECT * FROM T2 ORDER BY C2
```

Example 11: Retrieve all rows of table T1 with no specific ordering) and connect the result table to the rows of table T2, which have been ordered by the first column of table T2. The ORDER BY ORDER OF clause in the fullselect specifies that the order of the rows in the result table of the union is to be inherited by the final result.

```
SELECT *
  FROM (SELECT * FROM T1
         UNION ALL
        (SELECT * FROM T2 ORDER BY 1)
       ) AS UTABLE
  ORDER BY ORDER OF UTABLE;
```

Example 12: The following example uses a query to join data from a table to the result table of a nested table expression. The query uses the ORDER BY ORDER OF clause to order the rows of the result table using the order of the rows of the nested table expression.

```
SELECT T1.C1, T1.C2, TEMP.Cy, TEMP.Cx
  FROM T1,
    (SELECT T2.C1, T2.C2 FROM T2 ORDER BY 2) AS TEMP(Cx, Cy)
  WHERE Cy = T1.C1
  ORDER BY ORDER OF TEMP;
```

Example 13: Using the EMP_ACT table, find the project numbers that have an employee whose salary is in the top three salaries for all employees.

```
SELECT EMP_ACT.EMPNO, PROJNO
  FROM EMP_ACT
  WHERE EMP_ACT.EMPNO IN
    (SELECT EMPLOYEE.EMPNO
     FROM EMPLOYEE
     ORDER BY SALARY DESC
     FETCH FIRST 3 ROWS ONLY);
```

Example 14: Assume that an external function named ADDYEARS exists. For a given date, the function adds a given number of years and returns a new date. (The data types of the two input parameters to the function are DATE and INTEGER.) Get the employee number and employee name for all employees who have been hired within the last 5 years.

```
SELECT EMPNO, FIRSTNME, LASTNAME
  FROM DSN8B10.EMP
  WHERE ADDYEARS(HIREDATE, 5) > CURRENT DATE;
```

To distinguish the different types of joins, to show nested table expressions, and to demonstrate how to combine join columns, the remaining examples use these two tables:

```
The PARTS table                        The PRODUCTS table
PART      PROD#   SUPPLIER             PROD#     PRODUCT        PRICE
=======   =====   ============         =====     ===========    =====
WIRE      10      ACWF                 505       SCREWDRIVER    3.70
OIL       160     WESTERN_CHEM         30        RELAY          7.55
MAGNETS   10      BATEMAN              205       SAW            18.90
PLASTIC   30      PLASTIK_CORP         10        GENERATOR      45.75
BLADES    205     ACE_STEEL
```

Example 15: Join the tables on the PROD# column to get a table of parts with their suppliers and the products that use the parts:

```
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
   FROM PARTS, PRODUCTS
   WHERE PARTS.PROD# = PRODUCTS.PROD#;
```

or

```
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
  FROM PARTS INNER JOIN PRODUCTS
    ON PARTS.PROD# = PRODUCTS.PROD#;
```

Either one of these two statements give this result:

```
PART        SUPPLIER       PROD#     PRODUCT
=======     ============   =====     ==========
WIRE        ACWF           10        GENERATOR
MAGNETS     BATEMAN        10        GENERATOR
PLASTIC     PLASTIK_CORP   30        RELAY
BLADES      ACE_STEEL      205       SAW
```

Notice two things about the example:

- There is a part in the parts table (OIL) whose product (#160) is not listed in the products table. There is a product (SCREWDRIVER, #505) that has no parts listed in the parts table. Neither OIL nor SCREWDRIVER appears in the result of the join.

  An *outer join*, however, includes rows where the values in the joined columns do not match.
- There is explicit syntax to express that this familiar join is not an outer join but an inner join. You can use INNER JOIN in the FROM clause instead of the comma. Use ON when you explicitly join tables in the FROM clause.

You can specify more complicated join conditions to obtain different sets of results. For example, eliminate the suppliers that begin with the letter A from the table of parts, suppliers, product numbers and products:

```
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
  FROM PARTS INNER JOIN PRODUCTS
    ON PARTS.PROD# = PRODUCTS.PROD#
    AND SUPPLIER NOT LIKE 'A%';
```

The result of the query is all rows that do not have a supplier that begins with A:

```
PART        SUPPLIER       PROD#     PRODUCT
=======     ============   =====     ==========
MAGNETS     BATEMAN        10        GENERATOR
PLASTIC     PLASTIK_CORP   30        RELAY
```

Example 16: Join the tables on the PROD# column to get a table of all parts and products, showing the supplier information, if any.

```
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
  FROM PARTS FULL OUTER JOIN PRODUCTS
    ON PARTS.PROD# = PRODUCTS.PROD#;
```

The result is:

```
PART        SUPPLIER       PROD#     PRODUCT
=======     ============   =====     ==========
WIRE        ACWF           10        GENERATOR
MAGNETS     BATEMAN        10        GENERATOR
PLASTIC     PLASTIK_CORP   30        RELAY
BLADES      ACE_STEEL      205       SAW
OIL         WESTERN_CHEM   160       (null)
(null)      (null)         (null)    SCREWDRIVER
```

The clause FULL OUTER JOIN includes unmatched rows from both tables. Missing values in a row of the result table are filled with nulls.

Example 17: Join the tables on the PROD# column to get a table of all parts, showing what products, if any, the parts are used in:

```
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
  FROM PARTS LEFT OUTER JOIN PRODUCTS
    ON PARTS.PROD# = PRODUCTS.PROD#;
```

The result is:

```
PART        SUPPLIER       PROD#     PRODUCT
=======     ============   =====     ==========
WIRE        ACWF           10        GENERATOR
MAGNETS     BATEMAN        10        GENERATOR
PLASTIC     PLASTIK_CORP   30        RELAY
BLADES      ACE_STEEL      205       SAW
OIL         WESTERN_CHEM   160       (null)
```

The clause LEFT OUTER JOIN includes rows from the table identified before it where the values in the joined columns are not matched by values in the joined columns of the table identified after it.

Example 18: Join the tables on the PROD# column to get a table of all products, showing the parts used in that product, if any, and the supplier.

```
SELECT PART, SUPPLIER, PRODUCTS.PROD#, PRODUCT
  FROM PARTS RIGHT OUTER JOIN PRODUCTS
    ON PARTS.PROD# = PRODUCTS.PROD#;
```

The result is:

```
PART        SUPPLIER       PROD#     PRODUCT
=======     ============   =====     ===========
WIRE        ACWF           10        GENERATOR
MAGNETS     BATEMAN        10        GENERATOR
PLASTIC     PLASTIK_CORP   30        RELAY
BLADES      ACE_STEEL      205       SAW
(null)      (null)         505       SCREWDRIVER
```

The clause RIGHT OUTER JOIN includes rows from the table identified after it where the values in the joined columns are not matched by values in the joined columns of the table identified before it.

Example 19: The result of Example 16 (a full outer join) shows the product number for SCREWDRIVER as null, even though the PRODUCTS table contains a product number for it. This is because PRODUCTS.PROD# was not listed in the SELECT list of the query. Revise the query using COALESCE so that all part numbers from both tables are shown.

```
SELECT PART, SUPPLIER,
      COALESCE(PARTS.PROD#, PRODUCTS.PROD#) AS PRODNUM, PRODUCT
      FROM PARTS FULL OUTER JOIN PRODUCTS
        ON PARTS.PROD# = PRODUCTS.PROD#;
```

In the result, notice that the AS clause (AS PRODNUM), provides a name for the result of the COALESCE function:

```
PART        SUPPLIER       PRODNUM   PRODUCT
=======     ============   =======   ===========
WIRE        ACWF           10        GENERATOR
MAGNETS     BATEMAN        10        GENERATOR
PLASTIC     PLASTIK_CORP   30        RELAY
BLADES      ACE_STEEL      205       SAW
OIL         WESTERN_CHEM   160       (null)
(null)      (null)         505       SCREWDRIVER
```

Example 20: For all parts that are used in product numbers less than 200, show the part, the part supplier, the product number, and the product name. Use a nested table expression.

```
SELECT PART, SUPPLIER, PRODNUM, PRODUCT
  FROM (SELECT PART, PROD# AS PRODNUM, SUPPLIER
                FROM PARTS
                   WHERE PROD# < 200) AS PARTX
            LEFT OUTER JOIN PRODUCTS
                ON PRODNUM = PROD#;
```

The result is:

```
PART        SUPPLIER         PRODNUM     PRODUCT
=======     ============     =======     ==========
WIRE        ACWF             10          GENERATOR
MAGNETS     BATEMAN          10          GENERATOR
PLASTIC     PLASTIK_CORP     30          RELAY
OIL         WESTERN_CHEM     160         (null)
```

Example 21: Examples of statements with DISTINCT specified more than once in a subselect:

```
SELECT DISTINCT COUNT(DISTINCT A1), COUNT(A2)
  FROM T1;

SELECT COUNT(DISTINCT A))
  FROM T1
  WHERE A3 > 0
  HAVING AVG(DISTINCT A4) >1;
```

Example 22: **Examples of cross join to combine information for all customers with all states.**

Use a cross join to combine information for all customers with all of the states. The cross join combines all rows in both tables and creates a Cartesian product. Assume that the following tables exist:

```
Customer:
---------------------
ACOL1     |  ACOL2
```

```
--------------------
A1        |  AA1
A2        |  AA2
A3        |  AA3
--------------------

States:
--------------------
BCOL1     |  BCOL2
--------------------
B1        |  BB1
B2        |  BB2
--------------------
```

The following two select statements produce identical results:

```
SELECT * FROM customer CROSS JOIN states

SELECT * FROM A, B
```

The result table for either of these select statements looks like the following:

```
-----------------------------------------
ACOL1     |  ACOL2  |  BCOL1   |  BCOL2
-----------------------------------------
A1        |  AA1    |  B1      |  BB1
A1        |  AA1    |  B2      |  BB2
A2        |  AA2    |  B1      |  BB1
A2        |  AA2    |  B2      |  BB2
A3        |  AA3    |  B1      |  BB1
A3        |  AA3    |  B2      |  BB2
-----------------------------------------
```

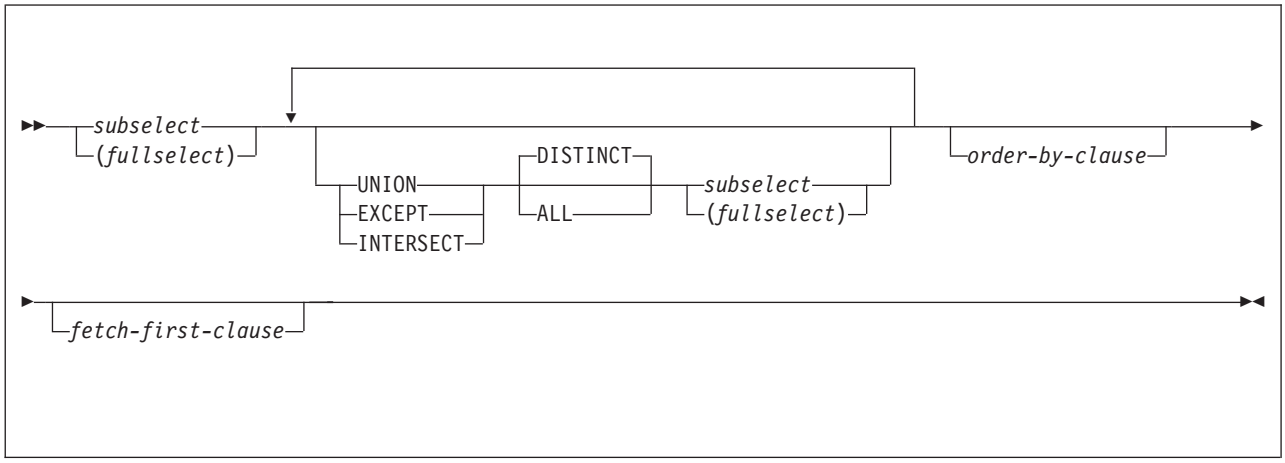
Example 22: **Example of using a *typed-correlation-clause* when referencing a generic table function.**

In the following select statement, 'tf6' is a generic table function defined using the CREATE FUNCTION (external table) statement. The *typed-correlation-clause* is used to define the column names and data types of the result table.

```
SELECT c1, c2
 FROM T1(tf6('abcd'))
  AS z (c1 int, c2 varchar(100));
```

fullselect

The fullselect is a component of the *select-statement*, ALTER TABLE statement for the definition of a materialized query table, CREATE TABLE statement, CREATE VIEW statement, DECLARE GLOBAL TEMPORARY TABLE statement, and INSERT statement.



A fullselect that is enclosed in parentheses is called a *subquery*. For example, a subquery can be used in a search condition.

A *scalar-fullselect* is a fullselect, enclosed in parentheses, that returns a single result row and a single result column. If the result of the fullselect is no rows, then the null value is returned. An error is returned if there is more than one row in the result. For example, a scalar-fullselect can be used in the assignment clause of the DELETE, UPDATE and MERGE statements.

A *row-fullselect* is a fullselect that returns a single row. An error is returned if there is more than one row in the result. For example, a row-fullselect can be used in the assignment clause of the DELETE and UPDATE statements.

UNION, EXCEPT, or INTERSECT

The set operators, UNION, EXCEPT, and INTERSECT, correspond to the relational operators union, difference, and intersection. A *fullselect* specifies a result table. If a set operator is not used, the result of the fullselect is the result of the specified subselect. Otherwise, the result table is derived by combining the two other result tables (R1 and R2) subject to the specified set operator.

UNION DISTINCT or UNION ALL

If UNION ALL is specified, the result consists of all rows in R1 and R2. With UNION DISTINCT, the result is the set of all rows in either R1 or R2 with the redundant duplicate rows eliminated. In either case, each row of the result table of the union is either a row from R1 or a row from R2.

The expression that corresponds to the *n*th column in R1 and R2 can reference columns with column masks. The *n*th column of the result of the union can be derived from the masked values in R1 or R2.

With UNION DISTINCT, the elimination of the duplicate rows is based on the unmasked values in R1 and R2. Because all rows are from R1 or R2, the output values in the result table of the union may vary when one or more of the following conditions occur:

- The expression corresponding to the *n*th column in R1 references columns with column masks, but the expression corresponding to the *n*th column in R2 does not, or vise versa.
- The expressions corresponding to the *n*th column in R1 and R2 reference columns with different column masks.
- The column mask definition references columns that are not the same target column for which the column mask is defined, and those columns are not part of the UNION DISTINCT operation. It is recommended that the column mask definition does not reference other columns from the target table.

For example, a row in R1 is derived from the masked value, and a row in R2 is derived from the unmasked value. If the row in the result table is from R1, the masked value is returned. If the row in the result table is from R2, the unmasked value is returned.

EXCEPT and INTERSECT can be intermixed with UNION if the rows in R1 and R2 for EXCEPT and INTERSECT do not reference columns with column masks

For compatibility with other SQL implementations, UNIQUE can be specified as a synonym for DISTINCT.

EXCEPT DISTINCT or EXCEPT ALL

If EXCEPT ALL is specified, the result consists of all rows from only R1, including significant redundant duplicate rows. With EXCEPT DISTINCT, the result consists of all rows that are only in R1, with redundant duplicate rows eliminated. In either case, each row in the result table of the difference is a row from R1 that does not have a matching row in R2.

Column masks cannot be applied to the select lists that derive the final result table of set operations if any of the set operators that are used to derive the final result table is EXCEPT ALL or EXCEPT DISTINCT.

For compatibility with other SQL implementations, MINUS can be specified as a synonym for EXCEPT, and UNIQUE can be specified as a synonym for DISTINCT.

INTERSECT DISTINCT or INTERSECT ALL

If INTERSECT ALL is specified, the result consists of all rows that are both in R1 and R2, including significant redundant duplicate rows. With INTERSECT DISTINCT, the result consists of all rows that are in both R1 and R2, with redundant duplicate rows eliminated. In either case each row of the result table of the intersection is a row that exists in both R1 and R2.

Column masks cannot be applied to the select lists that derive the final result table of set operations if any of the set operators that are used to derive the final result table is INTERSECT ALL or INTERSECT DISTINCT.

For compatibility with other SQL implementations, UNIQUE can be specified as a synonym for DISTINCT.

Rules for columns:

- R1 and R2 must have the same number of columns, and the data type of the *n*th column of R1 must be compatible with the data type of the *n*th column of R2.
- The nth column of the result of a set operator is derived from the nth columns of R1 and R2. The attributes of the result columns are determined using the rules for result columns.

- R1 and R2 must not include columns having a data type of CLOB, BLOB, DBCLOB, XML, or a distinct type that is based on any of these types. However, this rule is not applicable when UNION ALL is used with the set operator.
- If the *n*th column of R1 and the *n*th column of R2 have the same result column name, the *n*th column of the result table of the set operation has the same result column name. Otherwise, the *n*th column of the result table of the set operation is unnamed.
- Qualified column names cannot be used in the ORDER BY clause when the set operators are specified.

For information on the valid combinations of operand columns and the data type of the result column, see "Rules for result data types" on page 144.

Duplicate rows: Two rows are duplicates if the value in each column in the first row is equal to the corresponding value of the second row. For determining duplicates, two null values are considered equal.

The DECFLOAT data type allows for multiple bit representations of the same number. For example 2.00 and 2.0 are two numbers with the same coefficient, but different exponent values. See "Numeric comparisons" on page 134 section for more information. So if the result table of UNION contains a DECFLOAT column and multiple bit representations of the same number exist, the one returned is unpredictable.

Operator precedence: When multiple set operations are combined in an expression, set operations within parentheses are performed first. If the order is not specified by parentheses, set operations are performed from left to right with the exception that all INTERSECT operations are performed before any UNION or any EXCEPT operations.

Results of set operators: The following table illustrates the results of all set operations, with rows from result table R1 and R2 as the first two columns and the result of each operation on R1 and R2 under the corresponding column heading.

Table 91. Example of UNION, EXCEPT, and INTERSECT set operations on result tables R1 and R2.

Rows in R1	Rows in R2	Result of UNION ALL	Result of UNION DISTINCT	Result of EXCEPT ALL	Result of EXCEPT DISTINCT	Result of INTERSECT ALL	Result of INTERSECT DISTINCT
1	1	1	1	1	2	1	1
1	1	1	2	2	5	1	3
1	3	1	3	2		3	4
2	3	1	4	2		4	
2	3	1	5	4			
2	3	2		5			
3	4	2					
4		2					
4		3					
5		3					
		3					
		3					
		3					

Table 91. Example of UNION, EXCEPT, and INTERSECT set operations on result tables R1 and R2. (continued)

Rows in R1	Rows in R2	Result of UNION ALL	Result of UNION DISTINCT	Result of EXCEPT ALL	Result of EXCEPT DISTINCT	Result of INTERSECT ALL	Result of INTERSECT DISTINCT
		4					
		4					
		4					
		5					

Examples of fullselects

Example 1: A query specifies the union of result tables R1 and R2. A column in R1 has the data type CHAR(10) and the subtype BIT. The corresponding column in R2 has the data type CHAR(15) and the subtype SBCS. Hence, the column in the union has the data type CHAR(15) and the subtype BIT. Values from the first column are converted to CHAR(15) by adding five trailing blanks.

Example 2: Show all the rows from DSN8B10.EMP.

```
SELECT * FROM DSN8B10.EMP;
```

Example 3: Using sample tables DSN8B10.EMP and DSN8B10.EMPPROJACT, list the employee numbers of all employees for which either of the following statements are true:

- Their department numbers begin with 'D'.
- They are assigned to projects whose project numbers begin with 'AD'.

```
SELECT EMPNO FROM DSN8B10.EMP
  WHERE WORKDEPT LIKE 'D%'
  UNION
SELECT EMPNO FROM DSN8B10.EMPPROJACT
  WHERE PROJNO LIKE 'AD
```

The result is the union of two result tables, one formed from the sample table DSN8B10.EMP, the other formed from the sample table DSN8B10.EMPPROJACT. The result—a one-column table—is a list of employee numbers. Because UNION, rather than UNION ALL, was used, the entries in the list are distinct. If instead UNION ALL were used, certain employee numbers would appear in the list more than once. These would be the numbers for employees in departments that begin with 'D' while their projects begin with 'AD'.

Example 4: Specify a series of unions and order the results by the first column of the final result table.

```
SELECT * FROM T1
UNION
SELECT * FROM T2
UNION
SELECT * FROM T3
ORDER BY 1;
```

Example 5: Specify a series of unions and order the results by the first column of the final result table. The first ORDER BY clause order the rows of the result of the first union by the first column of that result table. The second ORDER BY clause is applied as part of the outer fullselect and it causes the rows of the final result table to be ordered by the first column of the final result table.

```
(SELECT * FROM T1
 UNION
 SELECT * FROM T2
 ORDER BY 1)
UNION
SELECT * FROM T3
ORDER BY 1;
```

Example 6: Assume that tables T1 and T2 exist and each contain the same number of columns named C1, C2, and so on. This example of the EXCEPT operator produces all rows that are in T1 but not in T2, with duplicate rows removed:

```
(SELECT * FROM T1)
  EXCEPT DISTINCT
(SELECT * FROM T2)
```

Example 7: Assume that tables T1 and T2 exist and each contain the same number of columns named C1, C2, and so on. This example of the INTERSECT operator produces all rows that are in both table T1 and table T2, with duplicate rows removed:

```
(SELECT * FROM T1)
  INTERSECT DISTINCT
(SELECT * FROM T2)
```

Character conversion in set operations and concatenations

The SQL operations that combine strings include concatenation, set operators, and the IN list of an IN predicate. Within an SQL statement, concatenation combines two or more strings into a new string. Within a fullselect, set operation, or the IN list of an IN predicate combine two or more string columns resulting from the subselects into results column.

All such operations have the following in common:

- The choice of a result CCSID for the string or column
- The possible conversion of one or more of the component strings or columns to the result CCSID

For all such operations, the rules for those two actions are the same, as described in "Selecting the result CCSID" on page 817. These rules also apply to the COALESCE scalar function.

Selecting the result CCSID

The result CCSID is selected at package prepare time. The result CCSID is the CCSID of one of the operands.

Two operands: When two operands are used, the result CCSID is determined by the operand types, their CCSIDs, and their relative positions in the operation. When a CCSID is X'FFFF', the result CCSID is always X'FFFF', and no character conversions take place. When neither CCSID is X'FFFF', the rules for selecting the result CCSID are identical to the ones for string comparison. See "String comparisons" in Chapter 3, “Functions,” on page 337.

Three or more operands:

If all the operands have the same CCSID, the result CCSID is the common CCSID.

If at least one of the CCSIDs has the value X'FFFF', the result CCSID also has the value X'FFFF'.

Otherwise, selection proceeds as follows:

1. The rules for a pair of operands are applied to the first two operands. This picks a “candidate” for the second step. The candidate is the operand that would furnish the result CCSID if just the first two operands were involved in the operation.
2. The rules are applied to the Step 1 candidate and the third operand, thereby selecting a second candidate.
3. If a fourth operand is involved, the rules are applied to the second candidate and fourth operand, to select a third candidate, and so on.

The process continues until all operands have been used. The remaining candidate is the one that furnishes the result CCSID. Whenever the rules for a pair are applied to a candidate and an operand, the candidate is considered to be the first operand.

Consider, for example, the following concatenation:

```
A CONCAT B CONCAT C
```

Here, the rules are first applied to the strings A and B. Suppose that the string selected as candidate is A. Then the rules are applied to A and C. If the string selected is again A, then A furnishes the result CCSID. Otherwise, C furnishes the result CCSID.

Character conversion of components: An operand of concatenation or the selected argument of the COALESCE scalar function is converted, if necessary, to the coded character set of the result string. Each string of an operand of a set operation is converted, if necessary, to the coded character set of the result column. In either case, the coded character set is the one identified by the result CCSID. Character conversion is necessary only if all of the following are true:

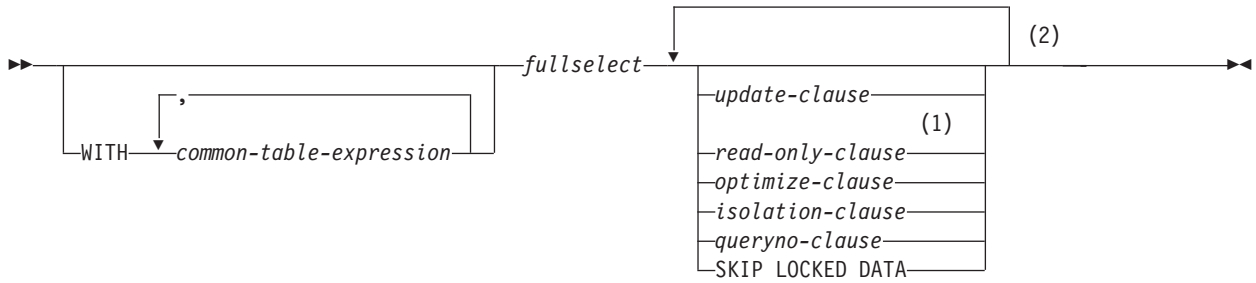
- The result and operand CCSIDs are different.
- Neither CCSID is X'FFFF' (neither string is defined as BIT data).
- The string is neither null nor empty.
- The SYSSTRINGS catalog table indicates that conversion is necessary.

An error occurs if a character of a string cannot be converted, SYSSTRINGS is used but contains no information about the CCSID pair, or DB2 cannot do the

conversion through z/OS support for Unicode. A warning occurs if a character of a string is converted to the substitution character.

select-statement

The *select-statement* is the form of a query that can be directly specified in a DECLARE CURSOR statement or FOR statement, prepared and then referenced in a DECLARE CURSOR statement, or directly specified in an SQLJ assignment clause. It can also be issued using SPUFI or the command line processor which causes a result table to be displayed at your terminal. In any case, the result table specified by a *select-statement* is the result of the *fullselect*.



Notes:

1 The *read-only-clause* must not be specified if *update-clause* is specified.

2 The same clause must not be specified more than one time.

The tables and view identified in a select statement can be at the current server or any DB2 subsystem with which the current server can establish a connection.

For local queries on DB2 for z/OS or remote queries in which the server and requester are DB2 for z/OS, if a table is encoded as ASCII or Unicode, the retrieved data is encoded in EBCDIC. For information on retrieving data encoded in ASCII or Unicode, see *DB2 Application Programming and SQL Guide*.

A select statement can implicitly or explicitly invoke user-defined functions or implicitly invoke stored procedures. This technique is known as *nesting* of SQL statements. A function or procedure is implicitly invoked in a select statement when it is invoked at a lower level. For instance, if you invoke a user-defined function from a select statement and the user-defined function invokes a stored procedure, you are implicitly invoking the stored procedure. When a SELECT statement refers to a table, any SQL statements that are implicitly invoked (as a result of nested functions or procedures) must not result in an SQL data change statement that modifies the same table.

For example, suppose that you execute this SQL statement at level 1 of nesting:

```
SELECT UDF1(C1) FROM T1;
```

You cannot execute this SQL statement at a lower level of nesting:

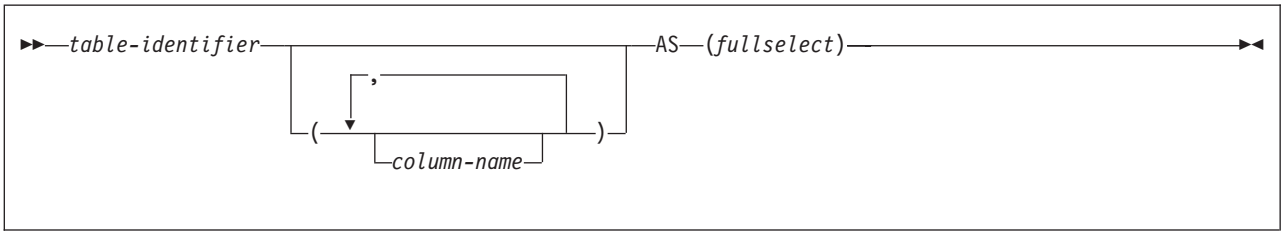
```
INSERT INTO T1 VALUES(...);
```

common-table-expression

A *common table expression* defines a result table with *table-identifier* that can be referenced in any FROM clause of the *fullselect* that follows.

Multiple common table expressions can be specified following the single WITH keyword. Each specified common table expression can also be referenced by name in the FROM clause of subsequent common table expressions.

common-table-expression



If a list of columns is specified, it must consist of as many names as there are columns in the result table of the fullselect. Each *column-name* must be unique and unqualified. If these column names are not specified, the names are derived from the select list of the *fullselect* used to define the common table expression.

table-identifier must be an unqualified SQL identifier, and it must be different from any other *table-identifier* in the same statement. If the common table expression is specified in an INSERT statement, the *table-identifier* must not be the same as the table or view name that is the object of the insert. If the common table expression is specified in a CREATE VIEW statement, the *table-identifier* must not be the same as the view name that is created. A common table expression *table-identifier* can be specified as a table name in any FROM clause throughout the *fullselect*.

If more than one common table expression is defined in the same statement, cyclic references between the common table expressions are not permitted. A *cyclic reference* occurs when two common table expressions *dt1* and *dt2* are created such that *dt1* refers to *dt2* and *dt2* refers to *dt1*. Furthermore, a common table expression defined before cannot refer to subsequent common table expressions.

A common table expression name can only be referenced in the *select-statement*, SELECT INTO statement, INSERT statement, CREATE VIEW statement, or RETURN statement that defines it.

If a *select-statement*, SELECT INTO statement, INSERT statement, or CREATE VIEW statement that is not contained in a trigger definition refers to a unqualified table name, the following rules are applied to determine which table is actually being referenced:

- If the unqualified name corresponds to one or more common table expression names that are specified in the *select-statement*, the name identifies the common table expression that is in the innermost scope.
- Otherwise, the name identifies a persistent table, a temporary table, or a view that is present in the default schema.

If a *select-statement*, SELECT INTO statement, INSERT statement, or CREATE VIEW statement that is contained in a trigger definition refers to a unqualified table name, the following rules are applied to determine which table is actually being referenced:

- If the unqualified name corresponds to one or more common table expression names that are specified in the *select-statement*, the name identifies the common table expression that is in the innermost scope.
- If the unqualified name corresponds to a transition table name, the name identifies that transition table.
- Otherwise, the name identifies a persistent table, a temporary table, or a view that is present in the default schema.

The common table expression is also optional prior to the fullselect in the CREATE VIEW and INSERT statements. However, the use of common table expressions is not allowed in a INSERT within SELECT statement.

A common table expression can be used:

- In place of a view to avoid creating the view (when general use of the view is not required and positioned updates or deletes are not used)
- When the result table that you want is based on host variables
- When the same result table needs to be shared in a *fullselect*
- When the result needs to be derived using recursion

If a *fullselect* of a common table expression contains a reference to itself in a FROM clause, the common table expression is a *recursive common table expression*. Queries using recursion are useful in supporting applications such as bill of materials (BOM), reservation systems, and network planning.

The following must be true of a recursive common table expression:

- Each *fullselect* that is part of the recursion cycle must start with SELECT or SELECT ALL. Use of SELECT DISTINCT is not allowed. Furthermore, the set operators must use the ALL keyword.
- The column names must be specified following the *table-name* of the common table expression.
- The first *fullselect* of the first set operator (the initialization *fullselect*) must not include a reference to the common table expression itself in any FROM clause).
- If a column name of the common table expression is referred to in the iterative *fullselect*, the data type, length, and CCSID for the column are determined based on the initialization *fullselect*. The corresponding column in the iterative *fullselect* must have the same data type and length as the data type and length determined based on the initialization *fullselect* and the CCSID must match. However, for character string types, the length of the two data types can differ. In this case, the column in the iterative *fullselect* must have a length that would always be assignable to the length determined from the initialization *fullselect*. If a column of a recursive common table expression is not used recursively in its definition, the data type, length, and CCSID for the column are determined by applying rules associated with non-recursive queries.
- Each *fullselect* that is part of the recursion cycle must not include any aggregate functions, GROUP BY clauses, or HAVING clauses. The FROM clauses of these *fullselects* can include at most one reference to a common table expression that is part of a recursion cycle.
- Subqueries (scalar or quantified) must not be part of any recursion cycles.

- Outer join must not be part of any recursion cycles.

When developing recursive common table expressions, remember that an infinite recursion cycle (loop) can be created. Check that recursion cycles will terminate. This is especially important if the data involved is cyclic. A recursive common table expression is expected to include a predicate that will prevent an infinite loop. The recursive common table expression is expected to include:

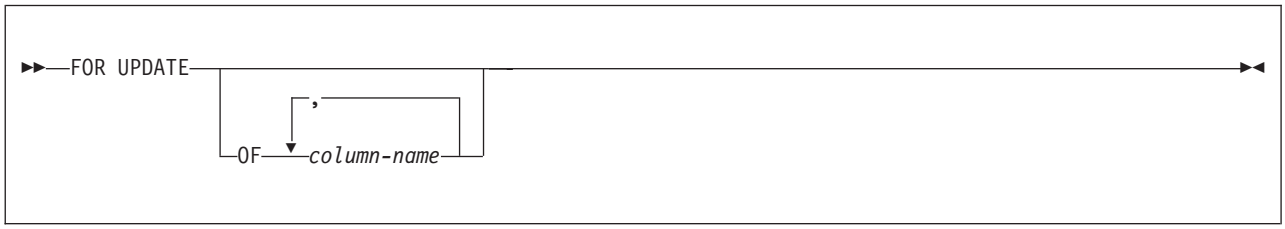
- In the iterative *fullselect*, an integer column incremented by a constant.
- A predicate in the WHERE clause of the iterative *fullselect* in the form of "*counter_col* < *constant*" or "*counter_col* < :*hostvar*". A warning is issued if this syntax is not found.

If the result of a recursive common table expression is used to derive the final result table, and if a column mask is used to mask the column values in the final result table, the column mask cannot be applied to a column that is specified in the *fullselect* of the recursive common table expression.

update-clause

The optional FOR UPDATE clause identifies the columns that can appear as targets in an assignment clause in a later positioned UPDATE statement.

update-clause



Each column name must be unqualified and must identify a column of the table or view identified in the first **FROM** clause of the fullselect. The clause must not be specified if the result table of the fullselect is read-only.

If **FOR UPDATE** clause is specified with a *column-name* list, and extended indicator variables are not enabled, *column-name* must be an updatable column.

If the **FOR UPDATE** clause is specified without a *column-name* list, the implicit list of column names is determined as follows:

- If extended indicator variables are enabled, all columns of the table or view that is identified in the first FROM clause of the fullselect
- If extended indicator variables are not enabled, all updatable columns of the table or view that is identified in the first FROM clause of the fullselect

If a dynamically prepared select-statement does not contain a positioned UPDATE clause, the cursor that is associated with the select statement cannot be referenced in a positioned UPDATE statement.

If a statically prepared *select-statement* does not contain an UPDATE clause and its result table is not read-only, an implicit UPDATE clause will result. The implicit list of column names is determined as follows:

- If extended indicator variables are enabled, all columns of the table or view that is identified in the first FROM clause of the fullselect
- If extended indicator variables are not enabled, all updatable columns of the table or view that is identified in the first FROM clause of the fullselect

The declaration of a cursor referred to in a positioned UPDATE statement need not include an **UPDATE** clause if the STDSQL(YES) or NOFOR SQL processing option is specified when the program is prepared. For more on the subject, see "Positioned updates of columns" on page 333.

When **FOR UPDATE** is used, FETCH operations referencing the cursor acquire U or X locks rather than S locks when:

- The isolation level of the statement is cursor stability.
- The isolation level of the statement is repeatable read or read stability and field U LOCK FOR RR/RS on installation panel DSNTIPI is set to get U locks.

- The isolation level of the statement is repeatable read or read stability and USE AND KEEP EXCLUSIVE LOCKS or USE AND KEEP UPDATE LOCKS is specified in the SQL statement, an X lock or a U lock, respectively, is acquired at fetch time.

No locks are acquired on declared temporary tables. For a discussion of U locks and S locks, see *DB2 Performance Monitoring and Tuning Guide*.

read-only-clause

The read-only clause specifies that the result table is read-only. Therefore, the cursor cannot be referred to in positioned UPDATE or DELETE statements.

read-only-clause

Some result tables are read-only by nature (for example, a table based on a read-only view.) FOR READ ONLY can still be specified for such tables, but the specification has no effect.

For tables in which updates and deletes are allowed, specifying FOR READ ONLY can possibly improve the performance of FETCH operations as DB2 can do blocking and avoid exclusive locks. For example, in programs that contain dynamic SQL statements without the FOR READ ONLY or ORDER BY clause, DB2 might open cursors as if the UPDATE clause was specified.

A read-only result table must not be referred to in an UPDATE or DELETE statement, whether it is read-only by nature or specified as FOR READ ONLY.

To take advantage of the possibly improved performance of FETCH operations while guaranteeing that selected data is not modified and preventing some types of deadlocks, you can specify FOR READ ONLY in combination with the optional syntax of USE AND KEEP ... LOCKS on the *isolation-clause*.

Alternative syntax and synonyms: FOR FETCH ONLY can be specified as a synonym for FOR READ ONLY.

Related concepts:

Block fetch (Introduction to DB2 for z/OS)

Problems with ambiguous cursors (DB2 Performance)

Related tasks:

Ensuring block fetch (DB2 Performance)

Related reference:

"FETCH" on page 1650

"UPDATE" on page 1933

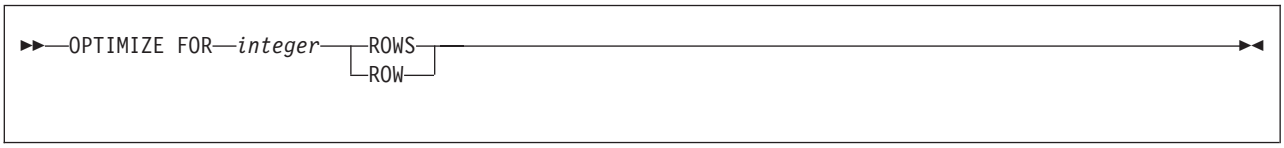
"DELETE" on page 1573

"isolation-clause" on page 827

optimize-clause

The OPTIMIZE clause requests special optimization of the *select-statement*.

optimize-clause



The *optimize-clause* tells DB2 to assume that the program does not intend to retrieve more than *integer* rows from the result table. Without this clause, DB2 assumes that all rows of the result table will be retrieved, unless the FETCH FIRST clause is specified. Optimizing for *integer* rows can improve performance. If this clause is omitted and the FETCH FIRST is specified, OPTIMIZE FOR *integer* ROWS is assumed, where *integer* is the value that is specified in the FETCH FIRST clause. DB2 will optimize the query based on the specified number of rows.

The clause does not limit the number of rows that can be fetched, change the result table, or change the order in which the rows are fetched. Any number of rows can be fetched, but performance can possibly degrade after *integer* fetches. In general, if you are retrieving only a few rows, specify OPTIMIZE FOR 1 ROW to influence the access path that DB2 selects. For more information about using this clause, see *DB2 Application Programming and SQL Guide*.

The value of *integer* must be a positive integer (not zero).

Row access controls indirectly affects the OPTIMIZE FOR clause because row access controls affect the rows that are accessible to the authorization ID or role of the subselect.

Column access controls do not affect the OPTIMIZE FOR clause.

Related tasks:

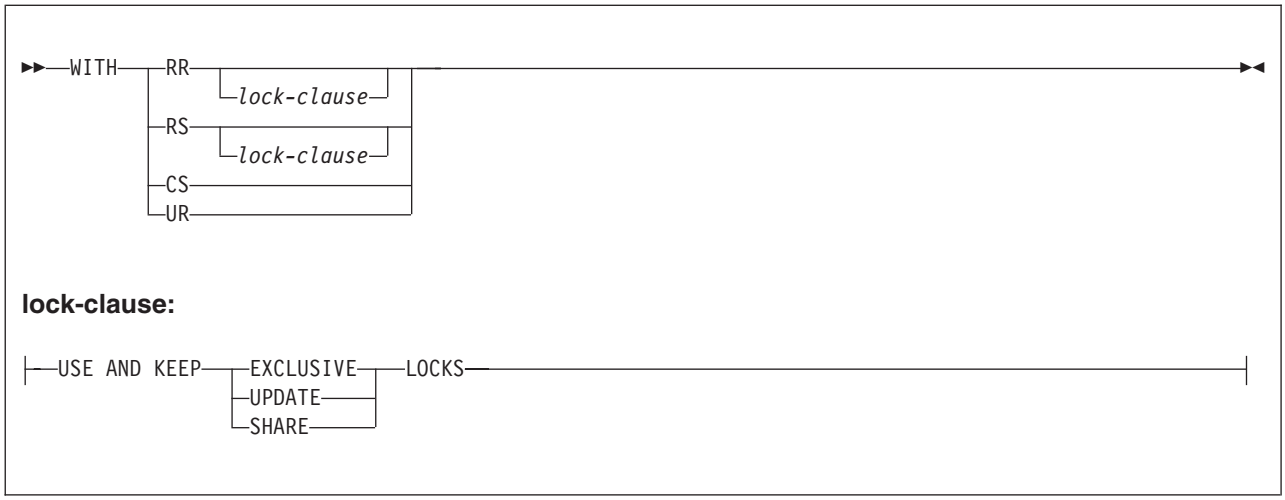
Minimizing the cost of retrieving few rows (DB2 Performance)

Optimizing retrieval for a small set of rows (DB2 Application programming and SQL)

isolation-clause

The *isolation-clause* specifies the isolation level at which the statement is executed. (Isolation level does not apply to declared temporary tables because no locks are acquired.)

isolation-clause



RR Repeatable read

RR *lock-clause*
Repeatable read, using and keeping the type of lock that is specified in *lock-clause* on all accessed pages and rows

RS Read stability

RS *lock-clause*
Read stability, using and keeping the type of lock that is specified in *lock-clause* on all accessed pages and rows

CS Cursor stability

UR Uncommitted read

lock-clause
Specifies the type of lock.

USE AND KEEP EXCLUSIVE LOCKS
USE AND KEEP UPDATE LOCKS
USE AND KEEP SHARE LOCKS
Specifies that DB2 is to acquire and hold X, U, or S locks, respectively.

WITH UR can be specified only if the result table of the fullselect or the SELECT INTO statement is read-only.

In an ODBC application, the SQLSetStmtAttr function can be used to set statement attributes that interact with the *lock-clause*. If SQLSetStmtAttr is invoked with a cursor's statement handle and specifying that its SQL_ATTR_CLOSE_BEHAVIOR is SQL_CC_RELEASE (locks are to be released when the cursor is closed), then irrespective of any *lock-clause*, lock used by the cursor that are not needed to protect the integrity of changed data are released..

Although requesting an UPDATE or EXCLUSIVE LOCK can reduce concurrency, it can prevent some types of deadlocks.

The **default** isolation level of the statement depends on:
- The isolation of the package or plan that the statement is bound in
- Whether the result table is read-only

Table 92 shows the default isolation level of the statement.

Table 92. Default isolation level based on the isolation level of the package or plan and whether the result table is read-only

If package isolation is:	And plan isolation is:	And the result table is:	Then the default isolation is:
RR	Any	Any	RR
RS	Any	Any	RS
CS	Any	Any	CS
UR	Any	Read-only	UR
		Not read-only	CS
Not specified	Not specified	Any	RR
	RR	Any	RR
	RS	Any	RS
	CS	Any	CS
	UR	Read-only	UR
		Not read-only	CS

A simple way to ensure that a result table is read-only is to specify FOR READ ONLY in the SQL statement.

Alternative syntax and synonyms: KEEP UPDATE LOCKS can be specified as a synonym for USE AND KEEP EXCLUSIVE LOCKS. However, KEEP UPDATE LOCKS can be specified only if FOR UPDATE OF is specified, and it is not supported in the SELECT INTO statement.

Related concepts:

Lock modes (DB2 Performance)

Related tasks:

Choosing an ISOLATION option (DB2 Performance)

Programming for concurrency (DB2 Performance)

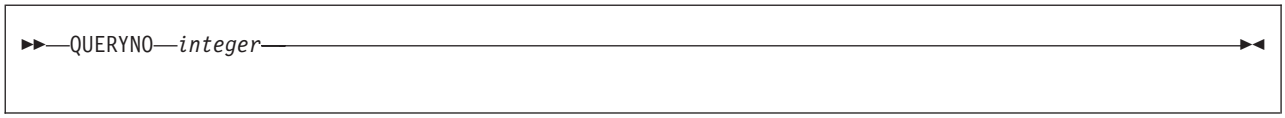
Related reference:

SQLSetStmtAttr() - Set statement attributes (DB2 Programming for ODBC)

queryno-clause

The QUERYNO clause specifies the number to be used for this SQL statement in EXPLAIN output and trace records. The number is used for the QUERYNO columns of the plan tables for the rows that contain information about this SQL statement. This number is also used in the QUERYNO column of the SYSIBM.SYSSTMT and SYSIBM.SYSPACKSTMT catalog tables.

queryno-clause



integer is the value to be used to identify this SQL statement in EXPLAIN output and trace records.

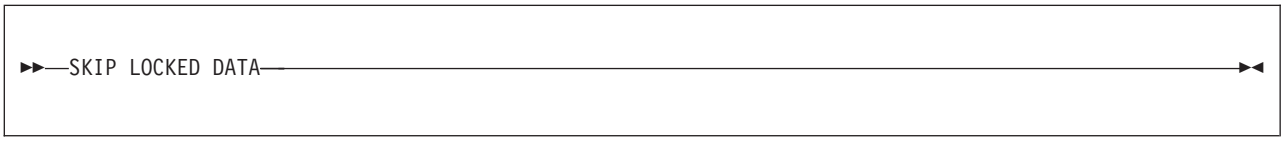
If the clause is omitted, the number associated with the SQL statement is the statement number assigned during precompilation. Thus, if the application program is changed and then precompiled, that statement number might change.

Using the QUERYNO clause to assign unique numbers to the SQL statements in a program is helpful:
- For simplifying the use of optimization hints for access path selection
- For correlating SQL statement text with EXPLAIN output in the plan table

For information on using optimization hints, such as enabling the system for optimization hints and setting valid hint values, and for information on accessing the plan table, see *DB2 Performance Monitoring and Tuning Guide*.

SKIP LOCKED DATA

The SKIP LOCKED DATA clause specifies that rows are skipped when incompatible locks that would block the progress of the statement are held on the rows by other transactions. These rows can belong to any accessed table that is specified in the statement. SKIP LOCKED DATA can be used only with isolation CS or RS and applies only to row level or page level locks.



Important: The recommendation is to not rely on the SKIP LOCKED DATA option to remove rows from results returned by a query. The SKIP LOCKED DATA option is meant only to prevent possibly incompatible locks from impeding the progress of queries that can tolerate possibly incomplete results. However, DB2 might use lock avoidance techniques to avoid taking certain locks.

SKIP LOCKED DATA is ignored if it is specified when the isolation level that is in effect is repeatable read (WITH RR) or uncommitted read (WITH UR). The default isolation level of the statement depends on the isolation level of the package or plan with which the statement is bound, and whether the result table is read-only.

Related concepts:

Lock avoidance (DB2 Performance)

Related tasks:

Improving concurrency for applications that tolerate incomplete results (DB2 Performance)

Related reference:

"select-statement" on page 819

"SELECT INTO" on page 1866

"UPDATE" on page 1933

"DELETE" on page 1573

"PREPARE" on page 1781

Examples of select statements

Examples of SELECT statements.

Example 1: Select all the rows from DSN8B10.EMP.

```
SELECT * FROM DSN8B10.EMP;
```

Example 2: Select all the rows from DSN8B10.EMP, arranging the result table in chronological order by date of hiring.

```
SELECT * FROM DSN8B10.EMP
   ORDER BY HIREDATE;
```

Example 3: Select the department number (WORKDEPT) and average departmental salary (SALARY) for all departments in the table DSN8B10.EMP. Arrange the result table in ascending order by average departmental salary.

```
SELECT WORKDEPT, AVG(SALARY)
  FROM DSN8B10.EMP
  GROUP BY WORKDEPT
  ORDER BY 2;
```

Example 4: Change various salaries, bonuses, and commissions in the table DSN8B10.EMP. Confine the changes to employees in departments D11 and D21. Use positioned updates to do this with a cursor named UP_CUR. Use a FOR UPDATE clause in the cursor declaration to indicate that all updatable columns are updated. Below is the declaration for a PL/I program.

```
EXEC SQL DECLARE UP_CUR CURSOR FOR
  SELECT WORKDEPT, EMPNO, SALARY, BONUS, COMM
    FROM DSN8B10.EMP
    WHERE WORKDEPT IN ('D11','D21')
    FOR UPDATE;
```

Beginning where the cursor is declared, all updatable columns would be updated. If only specific columns needed to be updated, such as only the salary column, the FOR UPDATE clause could be used to specify the salary column (FOR UPDATE OF SALARY).

Example 5: Find the maximum, minimum, and average bonus in the table DSN8B10.EMP. Execute the statement with uncommitted read isolation, regardless of the value of ISOLATION with which the plan or package containing the statement is bound. Assign 13 as the query number for the SELECT statement.

```
EXEC SQL
  SELECT MAX(BONUS), MIN(BONUS), AVG(BONUS)
    INTO :MAX, :MIN, :AVG
    FROM DSN8B10.EMP
    WITH UR
    QUERYNO 13;
```

If bind option EXPLAIN(YES) is specified, rows are inserted into the plan table. The value used for the QUERYNO column for these rows is 13.

Example 6: The cursor declaration shown below is in a PL/I program. In the query within the declaration, X.RMT_TAB is an alias for a table at some other DB2. Hence, when the query is used, it is processed using DRDA access. See “Distributed data” on page 35.

The declaration indicates that no positioned updates or deletes will be done with the query's cursor. It also specifies that the access path for the query be optimized for the retrieval of at most 50 rows. Even so, the program can retrieve more than

50 rows from the result table, which consists of the entire table identified by the alias. However, when more than 50 rows are retrieved, performance could possibly degrade.

```
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT * FROM X.RMT_TAB
  OPTIMIZE FOR 50 ROWS
  FOR READ ONLY;
```

The FETCH FIRST clause could be used instead of the OPTIMIZE FOR clause to ensure that only 50 rows are retrieved as in the following example:

```
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT * FROM X.RMT_TAB
  FETCH FIRST 50 ROWS ONLY;
```

Example 7: Assume that table DSN8B10.EMP has 1000 rows and you want to see the first five EMP_ROWID values that were inserted into DSN8B10.EMP_PHOTO_RESUME.

```
EXEC SQL DECLARE CS1 CURSOR FOR
  SELECT EMP_ROWID
    FROM FINAL TABLE (INSERT INTO DSN8B10.EMP_PHOTO_RESUME (EMPNO)
                      SELECT EMPNO FROM DSN8B10.EMP)
    FETCH FIRST 5 ROWS ONLY;
```

All 1000 rows are inserted into DSN8B10.EMP_PHOTO_RESUME, but only the first five are returned.

Related concepts:

How a SELECT statement works (Introduction to DB2 for z/OS)

Related tasks:

Coding SQL statements to avoid unnecessary processing (DB2 Performance)

Retrieving data by using the SELECT statement (DB2 Application programming and SQL)

Chapter 5. Statements

This section contains syntax diagrams, semantic descriptions, rules, and examples of the use of the SQL statements.

Table 93. SQL statements

SQL statement	Function	Topic
ALLOCATE CURSOR	Defines and associates a cursor with a result set locator variable	"ALLOCATE CURSOR" on page 847
ALTER DATABASE	Changes the description of a database	"ALTER DATABASE" on page 849
ALTER FUNCTION (external)	Changes the description of a user-defined external scalar or table function	"ALTER FUNCTION (external)" on page 852
ALTER FUNCTION (SQL scalar)	Changes the description of an SQL scalar function	"ALTER FUNCTION (SQL scalar)" on page 871
ALTER FUNCTION (SQL table)	Changes the description of an SQL table function	"ALTER FUNCTION (SQL table)" on page 899
ALTER INDEX	Changes the description of an index	"ALTER INDEX" on page 907
ALTER MASK	Changes the description of a column mask	"ALTER MASK" on page 926
ALTER PERMISSION	Changes the description of a row permission.	"ALTER PERMISSION" on page 928
ALTER PROCEDURE (external)	Changes the description of an external procedure	"ALTER PROCEDURE (external)" on page 930
ALTER PROCEDURE (SQL - external)	Changes the description of an external SQL procedure	"ALTER PROCEDURE (SQL - external)" on page 941
ALTER PROCEDURE (SQL - native)	Changes the description of or defines additional versions for a native SQL procedure	"ALTER PROCEDURE (SQL - native)" on page 947
ALTER SEQUENCE	Changes the description of a sequence	"ALTER SEQUENCE" on page 975
ALTER STOGROUP	Changes the description of a storage group	"ALTER STOGROUP" on page 981
ALTER TABLE	Changes the description of a table	"ALTER TABLE" on page 984
ALTER TABLESPACE	Changes the description of a table space	"ALTER TABLESPACE" on page 1074
ALTER TRUSTED CONTEXT	Changes the description of a trusted context	"ALTER TRUSTED CONTEXT" on page 1097
ALTER TRIGGER	Changes the description of a trigger.	"ALTER TRIGGER" on page 1094
ALTER VIEW	Regenerates a view	"ALTER VIEW" on page 1109
ASSOCIATE LOCATORS	Gets the result set locator value for each result set returned by a stored procedure	"ASSOCIATE LOCATORS" on page 1111
BEGIN DECLARE SECTION	Marks the beginning of a host variable declaration section	"BEGIN DECLARE SECTION" on page 1115
CALL	Calls a stored procedure	"CALL" on page 1117
CLOSE	Closes a cursor	"CLOSE" on page 1131

Table 93. SQL statements (continued)

SQL statement	Function	Topic
COMMENT	Replaces or adds a comment to the description of an object	"COMMENT" on page 1133
COMMIT	Ends a unit of recovery and commits the database changes made by that unit of recovery	"COMMIT" on page 1143
CONNECT	Connects the process to a server	"CONNECT" on page 1147
CREATE ALIAS	Defines an alias	"CREATE ALIAS" on page 1154
CREATE AUXILIARY TABLE	Defines an auxiliary table for storing LOB data	"CREATE AUXILIARY TABLE" on page 1158
CREATE DATABASE	Defines a database	"CREATE DATABASE" on page 1162
CREATE FUNCTION (external scalar)	Defines a user-defined external scalar function	"CREATE FUNCTION (external scalar)" on page 1166
CREATE FUNCTION (external table)	Defines a user-defined external table function	"CREATE FUNCTION (external table)" on page 1191
CREATE FUNCTION (sourced)	Defines a user-defined function that is based on an existing scalar or aggregate function	"CREATE FUNCTION (sourced)" on page 1210
CREATE FUNCTION (SQL scalar)	Defines a user-defined SQL scalar function	"CREATE FUNCTION (SQL scalar)" on page 1224
CREATE FUNCTION (SQL table)	Defines a user-defined SQL table function	"CREATE FUNCTION (SQL table)" on page 1251
CREATE GLOBAL TEMPORARY TABLE	Defines a created temporary table	"CREATE GLOBAL TEMPORARY TABLE" on page 1261
CREATE INDEX	Defines an index on a table	"CREATE INDEX" on page 1267
CREATE MASK	Defines a column mask	"CREATE MASK" on page 1299
CREATE PERMISSION	Defines a row permission.	"CREATE PERMISSION" on page 1310
CREATE PROCEDURE (external)	Defines an external stored procedure	"CREATE PROCEDURE (external)" on page 1319
CREATE PROCEDURE (SQL - external)	Defines an external SQL procedure	"CREATE PROCEDURE (SQL - external)" on page 1338
CREATE PROCEDURE (SQL - native)	Defines a native SQL procedure	"CREATE PROCEDURE (SQL - native)" on page 1350
CREATE ROLE	Defines a role	"CREATE ROLE" on page 1374
CREATE SEQUENCE	Defines a sequence	"CREATE SEQUENCE" on page 1375
CREATE STOGROUP	Defines a storage group	"CREATE STOGROUP" on page 1383
CREATE SYNONYM	Defines an alternate name for a table or view	"CREATE SYNONYM" on page 1386
CREATE TABLE	Defines a table	"CREATE TABLE" on page 1388

Table 93. SQL statements (continued)

SQL statement	Function	Topic
CREATE TABLESPACE	Defines a table space, which includes allocating and formatting the table space	"CREATE TABLESPACE" on page 1455
CREATE TRIGGER	Defines a trigger	"CREATE TRIGGER" on page 1482
CREATE TRUSTED CONTEXT	Defines a trusted context	"CREATE TRUSTED CONTEXT" on page 1500
CREATE TYPE	Defines a type (user-defined data type)	"CREATE TYPE (distinct)" on page 1516
CREATE VIEW	Defines a view of one or more tables or views	"CREATE VIEW" on page 1527
DECLARE CURSOR	Defines an SQL cursor	"DECLARE CURSOR" on page 1535
DECLARE GLOBAL TEMPORARY TABLE	Defines a declared temporary table	"DECLARE GLOBAL TEMPORARY TABLE" on page 1547
DECLARE STATEMENT	Declares names used to identify prepared SQL statements	"DECLARE STATEMENT" on page 1562
DECLARE TABLE	Provides the programmer and the precompiler with a description of a table or view	"DECLARE TABLE" on page 1563
DECLARE VARIABLE	Defines a CCSID for a host variable	"DECLARE VARIABLE" on page 1570
DELETE	Deletes one or more rows from a table	"DELETE" on page 1573
DESCRIBE CURSOR	Puts information about the result set associated with a cursor into a descriptor	"DESCRIBE CURSOR" on page 1591
DESCRIBE INPUT	Puts information about the input parameter markers of a prepared statement into a descriptor	"DESCRIBE INPUT" on page 1593
DESCRIBE OUTPUT	Describes the result columns of a prepared statement	"DESCRIBE OUTPUT" on page 1596
DESCRIBE PROCEDURE	Puts information about the result sets returned by a stored procedure into a descriptor	"DESCRIBE PROCEDURE" on page 1603
DESCRIBE TABLE	Describes the columns of a table or view	"DESCRIBE TABLE" on page 1606
DROP	Deletes objects	"DROP" on page 1609
END DECLARE SECTION	Marks the end of a host variable declaration section	"END DECLARE SECTION" on page 1631
EXCHANGE	Exchanges data between the specified base table and an associated clone table	"EXCHANGE" on page 1632
EXECUTE	Executes a prepared SQL statement	the EXECUTE statement
EXECUTE IMMEDIATE	Prepares and executes an SQL statement	"EXECUTE IMMEDIATE" on page 1639
EXPLAIN	Obtains information about how an SQL statement would be executed	"EXPLAIN" on page 1642
FETCH	Positions the cursor, returns data, or both positions the cursor and returns data	"FETCH" on page 1650
FREE LOCATOR	Removes the association between a LOB locator variable and its value	"FREE LOCATOR" on page 1678

Table 93. SQL statements (continued)

SQL statement	Function	Topic
GET DIAGNOSTICS	Provides diagnostic information about the last SQL statement that was executed	"GET DIAGNOSTICS" on page 1679
GRANT	The GRANT statement grants privileges to authorization IDs. There is a separate form of the statement for each of the classes of privilege.	"GRANT" on page 1695
GRANT (collection privileges)	Grants authority to create a package in a collection	"GRANT (collection privileges)" on page 1699
GRANT (database privileges)	Grants privileges on a database	"GRANT (database privileges)" on page 1700
GRANT (type or JAR file privileges)	Grants the usage privilege on a type (user-defined data type) or a JAR file	"GRANT (type or JAR file privileges)" on page 1725
GRANT (function or procedure privileges)	Grants privileges on a user-defined function or a stored procedure	"GRANT (function or procedure privileges)" on page 1703
GRANT (package privileges)	Grants authority to bind, execute, or copy a package	"GRANT (package privileges)" on page 1708
GRANT (plan privileges)	Grants authority to bind or execute an application plan	"GRANT (plan privileges)" on page 1711
GRANT (schema privileges)	Grants privileges on a schema	"GRANT (schema privileges)" on page 1712
GRANT (sequence privileges)	Grants privileges on a user-defined sequence	"GRANT (sequence privileges)" on page 1714
GRANT (system privileges)	Grants system privileges	"GRANT (system privileges)" on page 1715
GRANT (table or view privileges)	Grants privileges on a table or view	"GRANT (table or view privileges)" on page 1721
GRANT (use privileges)	Grants authority to use specified buffer pools, storage groups, or table spaces	"GRANT (use privileges)" on page 1728
HOLD LOCATOR	Allows a LOB locator variable to retain its association with its value beyond a unit of work	"HOLD LOCATOR" on page 1730
INCLUDE	Inserts declarations into a source program	"INCLUDE" on page 1732
INSERT	Inserts one or more rows into a table	"INSERT" on page 1734
LABEL	Replaces or adds a label on the description of a table, view, alias, or column	"LABEL" on page 1755
LOCK TABLE	Locks a table or table space partition in shared or exclusive mode	"LOCK TABLE" on page 1757
MERGE	Updates and/or inserts one or more rows of a table	"MERGE" on page 1760
OPEN	Opens a cursor	"OPEN" on page 1775
PREPARE	Prepares an SQL statement (with optional parameters) for execution	"PREPARE" on page 1781
REFRESH TABLE	Refreshes the data in a materialized query table	"REFRESH TABLE" on page 1803
RELEASE (connection)	Places one or more connections in the release pending status	"RELEASE (connection)" on page 1805
RELEASE SAVEPOINT	Releases a savepoint and any subsequently set savepoints within a unit of recovery	"RELEASE SAVEPOINT" on page 1807
RENAME	Renames an existing table or index	"RENAME" on page 1808

Table 93. SQL statements (continued)

SQL statement	Function	Topic
REVOKE	Revokes privileges from authorization IDs. There is a separate form of the statement for each of the classes of privilege	"REVOKE" on page 1812
REVOKE (collection privileges)	Revokes authority to create a package in a collection	"REVOKE (collection privileges)" on page 1819
REVOKE (database privileges)	Revokes privileges on a database	"REVOKE (database privileges)" on page 1821
REVOKE (type or JAR file privileges)	Revokes the usage privilege on a type (user-defined data type) or a JAR file	"REVOKE (type or JAR file privileges)" on page 1851
REVOKE (function or procedure privileges)	Revokes privileges on a user-defined function or a stored procedure	"REVOKE (function or procedure privileges)" on page 1824
REVOKE (package privileges)	Revokes authority to bind, execute, or copy a package	"REVOKE (package privileges)" on page 1831
REVOKE (plan privileges)	Revokes authority to bind or execute an application plan	"REVOKE (plan privileges)" on page 1834
REVOKE (schema privileges)	Revokes privileges on a schema	"REVOKE (schema privileges)" on page 1836
REVOKE (sequence privileges)	Revokes privileges on a user-defined sequence	"REVOKE (sequence privileges)" on page 1839
REVOKE (system privileges)	Revokes system privileges	"REVOKE (system privileges)" on page 1841
REVOKE (table or view privileges)	Revokes privileges on a table or view	"REVOKE (table or view privileges)" on page 1847
REVOKE (use privileges)	Revokes authority to use specified buffer pools, storage groups, or table spaces	"REVOKE (use privileges)" on page 1856
ROLLBACK	Ends a unit of recovery and backs out the changes to the database made by that unit of recovery, or partially rolls back the changes to a savepoint within the unit of recovery	"ROLLBACK" on page 1859
SAVEPOINT	Sets a savepoint within a unit of recovery	"SAVEPOINT" on page 1863
SELECT	Specifies the SELECT statement of the cursor	"SELECT" on page 1865
SELECT INTO	Specifies a result table of no more than one row and assigns the values to host variables	"SELECT INTO" on page 1866
SET CONNECTION	Establishes the database server of the process by identifying one of its existing connections	"SET CONNECTION" on page 1872
SET CURRENT APPLICATION ENCODING SCHEME	Assigns a value to the CURRENT APPLICATION ENCODING SCHEME special register	"SET CURRENT APPLICATION ENCODING SCHEME" on page 1883
SET CURRENT DEBUG MODE	Assigns a value to the CURRENT DEBUG MODE special register	"SET CURRENT DEBUG MODE" on page 1884
SET CURRENT DECFLOAT ROUNDING MODE	Assigns a value to the CURRENT DECFLOAT ROUNDING MODE special register	"SET CURRENT DECFLOAT ROUNDING MODE" on page 1886
SET CURRENT DEGREE	Assigns a value to the CURRENT DEGREE special register	"SET CURRENT DEGREE" on page 1889
SET CURRENT EXPLAIN MODE	Assigns a value to the CURRENT EXPLAIN MODE special register	"SET CURRENT EXPLAIN MODE" on page 1891

Table 93. SQL statements (continued)

SQL statement	Function	Topic
SET CURRENT LOCALE LC_CTYPE	Assigns a value to the CURRENT LOCALE LC_CTYPE special register	"SET CURRENT LOCALE LC_CTYPE" on page 1894
SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION	Assigns a value to the CURRENT MAINTAINED TABLE TYPES FOR MAINTAINED TABLE TYPES special register	"SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION" on page 1896
SET CURRENT OPTIMIZATION HINT	Assigns a value to the CURRENT OPTIMIZATION HINT special register	"SET CURRENT OPTIMIZATION HINT" on page 1898
SET CURRENT PACKAGE PATH	Assigns a value to the CURRENT PACKAGE PATH special register	"SET CURRENT PACKAGE PATH" on page 1899
SET CURRENT PACKAGESET	Assigns a value to the CURRENT PACKAGESET special register	"SET CURRENT PACKAGESET" on page 1903
SET CURRENT PRECISION	Assigns a value to the CURRENT PRECISION special register	"SET CURRENT PRECISION" on page 1905
SET CURRENT REFRESH AGE	Assigns a value to the CURRENT REFRESH AGE special register	"SET CURRENT REFRESH AGE" on page 1908
SET CURRENT ROUTINE VERSION	Assigns a value to the CURRENT ROUTINE VERSION special register	"SET CURRENT ROUTINE VERSION" on page 1910
SET CURRENT RULES	Assigns a value to the CURRENT RULES special register	"SET CURRENT RULES" on page 1912
SET CURRENT SQLID	Assigns a value to the CURRENT SQLID special register	"SET CURRENT SQLID" on page 1913
SET ENCRYPTION PASSWORD	Assign a value for the ENCRYPTION PASSWORD and an optional hint for the password	"SET ENCRYPTION PASSWORD" on page 1919
SET assignment-statement	Assigns values to variables and array elements	"SET assignment-statement" on page 1875
SET PATH	Assigns a value to the CURRENT PATH special register	"SET PATH" on page 1921
SET SCHEMA	Assigns a value to the CURRENT SCHEMA special register	"SET SCHEMA" on page 1924
SIGNAL	Signals an error or warning condition and optionally returns the specified message text	"SIGNAL statement" on page 2006
TRUNCATE	Deletes all rows from a table	"TRUNCATE" on page 1929
UPDATE	Updates the values of one or more columns in one or more rows of a table	"UPDATE" on page 1933
VALUES	Provides a way to invoke a user-defined function from a trigger	"VALUES" on page 1955
VALUES INTO	Assigns values to host variables	"VALUES INTO" on page 1956
WHENEVER	Defines actions to be taken on the basis of SQL return codes	"WHENEVER" on page 1961

How SQL statements are invoked

SQL statements are invoked in different ways depending on whether the statement is an executable or nonexecutable statement or the *select-statement*.

The SQL statements are classified as *executable* or *nonexecutable*. The description of each statement includes a heading on invocation that indicates whether or not the statement is executable.

Executable statements can be invoked in the following ways:

- Embedded in an application program
- Dynamically prepared and executed
- Dynamically prepared and executed using DB2 ODBC function calls
- Issued interactively

Depending on the statement, you can use some or all of these methods. The section on invocation in the description of each statement tells you which methods can be used.

A *nonexecutable statement* can only be embedded in an application program.

The *select-statement* is an additional SQL statement construct. (See "select-statement" on page 819.) It is used in a different way from other statements.

A *select-statement* can be invoked in the following ways:

- Included in DECLARE CURSOR and implicitly executed by OPEN
- Dynamically prepared, referred to in DECLARE CURSOR, and implicitly executed by OPEN
- Dynamically executed (no PREPARE required) using a DB2 ODBC function call
- Issued interactively

The first two methods are called, respectively, the *static* and the *dynamic* invocation of *select-statement*.

Embedding a statement in an application program

You can include SQL statements in a source program that will be submitted to the DB2 precompiler or coprocessor. Such statements are said to be *embedded* in the application program. An embedded statement can be placed anywhere in the application program where a host language statement is allowed. Each embedded statement must be preceded by a keyword (or keywords) to indicate that the statement is an SQL statement.

- In C and COBOL, each embedded statement must be preceded by the keywords EXEC SQL.
- In Java, each embedded statement must be preceded by the keywords #sql.
- In REXX, each embedded statement must be preceded by the keyword EXECSQL.

Executable statements: An executable statement embedded in an application program is executed every time a statement of the host language would be executed if specified in the same place. (Thus, for example, a statement within a loop is executed every time the loop is executed, and a statement within a conditional construct is executed only when the condition is satisfied.)

An embedded statement can contain references to host variables. A host variable referred to in this way can be used in one of two ways:

As *input*
The current value of the host variable is used in the execution of the statement.

As *output*
The variable is assigned a new value as a result of executing the statement.

In particular, all references to host variables in expressions and predicates are effectively replaced by current values of the variables; that is, the variables are used as input. The treatment of other references is described individually for each statement.

The successful or unsuccessful execution of the statement is indicated by setting the SQLCODE and SQLSTATE fields in the SQLCA.[23] You must therefore follow all executable statements by a test of SQLCODE or SQLSTATE. Alternatively, you can use the WHENEVER statement (which is itself nonexecutable) to change the flow of control immediately after the execution of an embedded statement.

Nonexecutable statements: An embedded nonexecutable statement is processed only by the precompiler or coprocessor. The precompiler or coprocessor reports any errors encountered in the statement. The statement is never executed, and acts as a no-operation if placed among executable statements of the application program. Therefore, do not follow such statements with a test of an SQL return code.

Dynamic preparation and execution

Your application program can dynamically build an SQL statement in the form of a character string placed in a host variable. In general, the statement is built from some data available to the application program (for example, input from a workstation).

In non-Java languages, the statement so constructed can be prepared for execution by means of the (embedded) statement PREPARE and executed by means of the (embedded) statement EXECUTE, as described in *DB2 Application Programming and SQL Guide*. Alternatively, you can use the (embedded) statement EXECUTE IMMEDIATE to prepare and execute a statement in one step. In Java, the statement can be prepared for execution by means of the Statement, PreparedStatement, and CallableStatement classes, and executed by means of their respective execute() methods.

The statement can also be prepared by calling the DB2 ODBC SQLPrepare function and then executed by calling the DB2 ODBC SQLExecute function. In both cases, the application does not contain an embedded PREPARE or EXECUTE statement. You can execute the statement, without preparation, by passing the statement to the DB2 ODBC SQLExecDirect function. *DB2 ODBC Guide and Reference* describes the APIs supported with this interface.

A statement that is going to be prepared must not contain references to host variables. It can instead contain parameter markers. (See Parameter markers in the description of the PREPARE statement for rules concerning parameter markers.) When the prepared statement is executed, the parameter markers are effectively replaced by current values of the host variables specified in the EXECUTE statement. (See the EXECUTE statement for rules concerning this replacement.)

23. SQLCODE and SQLSTATE cannot be in the SQLCA when the SQL processing option STDSQL(YES) is in effect. See "SQL standard language" on page 332.

After it is prepared, a statement can be executed several times with different values of host variables. Parameter markers are not allowed in the SQL statement prepared and executed using EXECUTE IMMEDIATE.

In non-Java languages, the successful or unsuccessful execution of the statement is indicated by the values returned in the SQLCODE and SQLSTATE fields in the SQLCA after the EXECUTE (or EXECUTE IMMEDIATE) statement. You should check the fields as described above for embedded statements. In Java, the successful or unsuccessful execution of the statement is handled by Java Exceptions.

As explained in "Authorization IDs and dynamic SQL" on page 75, the DYNAMICRULES behavior in effect determines the privilege set that is used for authorization checking when dynamic SQL statements are processed. The following table summarizes those privilege sets. (See Table 6 on page 75 for a list of the DYNAMICRULES bind option values that determine which behavior is in effect).

Table 94. DYNAMICRULES behaviors and authorization checking

DYNAMICRULES behavior	Privilege set
Run behavior	The union of the set of privileges held by each authorization ID of the process if the dynamically prepared statement is other than an ALTER, CREATE, DROP, GRANT, RENAME, or REVOKE statement. The privileges that are held by the SQL authorization ID of the process or the role of the primary authorization ID (if the process is running in a trusted context that is defined with the ROLE AS OBJECT OWNER clause), if the dynamic SQL statement is a CREATE, GRANT, or REVOKE statement.
Bind behavior	The privileges that are held by the primary authorization ID of the owner of the package or plan.
Define behavior	The privileges that are held by the authorization ID of the stored procedure or user-defined function owner (definer).
Invoke behavior	The privileges that are held by the authorization ID of the stored procedure or user-defined function invoker. However, if the invoker is the primary authorization ID of the process or the CURRENT SQLID value, secondary authorization IDs are also checked if they are needed for the required authorization. Therefore, in that case, the privilege set is the union of the set of privileges that are held by each authorization ID or role (if running in a trusted context).

Static invocation of a SELECT statement

A SELECT statement can be invoked statically in different ways.

You can include a SELECT statement as a part of the (nonexecutable) statement DECLARE CURSOR. Such a statement is executed every time you open the cursor by means of the (embedded) statement OPEN. After the cursor is open, you can retrieve the result table a row at a time by successive executions of the (embedded) SQL FETCH statement.

If the application is using DB2 ODBC, the SELECT statement is first prepared with the SQLPrepare function call. It is then executed with the SQLExecute function call. Data is then fetched with the SQLFetch function call. The application does not explicitly open the cursor.

The SELECT statement used in this way can contain references to host variables. These references are effectively replaced by the values that the variables have at the moment of executing OPEN.

The successful or unsuccessful execution of the SELECT statement is indicated by the values returned in the SQLCODE and SQLSTATE fields in the SQLCA after the OPEN. You should check the fields as described above for embedded statements.

If the application is using DB2 ODBC, the successful execution of the SELECT statement is indicated by the return code from the SQLExecute function call. If necessary, the application can retrieve the SQLCA by calling the SQLGetSQLCA function.

Dynamic invocation of a SELECT statement

Your application program can dynamically build a SELECT statement in the form of a character string placed in a host variable. In general, the statement is built from some data available to the application program (for example, a query obtained from a terminal).

The statement so constructed can be prepared for execution by means of the (embedded) statement PREPARE, and referred to by a (nonexecutable) statement DECLARE CURSOR. The statement is then executed every time you open the cursor by means of the (embedded) statement OPEN. After the cursor is open, you can retrieve the result table a row at a time by successive executions of the (embedded) SQL FETCH statement.

The SELECT statement used in that way must not contain references to host variables. It can instead contain parameter markers. (See “Notes” in “PREPARE” on page 1781 for rules concerning parameter markers.) The parameter markers are effectively replaced by the values of the host variables specified in the OPEN statement. (See “OPEN” on page 1775 for rules concerning this replacement.)

The successful or unsuccessful execution of the SELECT statement is indicated by the values returned in the SQLCODE and SQLSTATE fields in the SQLCA after the OPEN. You should check the fields as described above for embedded statements.

Interactive invocation

An SQL statement submitted to DB2 from a terminal is said to be issued interactively.

IBM relational database management systems allow you to enter SQL statements from a terminal. DB2 for z/OS provides SPUFI to prepare and execute SQL statements. Other products are also available. A statement entered in this way is said to be issued interactively.

A statement issued interactively must not contain parameter markers or references to host variables, because these make sense only in the context of an application program. For the same reason, there is no SQLCA involved.

SQL diagnostics information

DB2 uses a diagnostics area to store status information and diagnostics information about the execution of an executable SQL statement.

When an SQL statement other than GET DIAGNOSTICS or *compound-statement* is processed, the current diagnostics area is cleared before processing the SQL statement. As each SQL statement is processed, information about the execution of that SQL statement is recorded in the current diagnostics area as one or more completion conditions or exception conditions.

A completion condition indicates that the SQL statement completed successfully, completed with a warning condition, or completed with a not found condition. An exception condition indicates that the statement was not successful. The GET DIAGNOSTICS statement can be executed in most languages to return conditions and other information about the previously executed SQL statement from the diagnostics area. Additionally, the condition information is provided through language specific mechanisms for SQL procedures, and host language applications.

Related concepts:

"Detecting and processing error and warning conditions in host language applications"

Related reference:

"GET DIAGNOSTICS" on page 1679

"SQL-procedure-statement" on page 1968

Detecting and processing error and warning conditions in host language applications

Errors and warnings conditions in host language applications can be checked by using the SQLCODE or SQLSTATE host variables or by using the SQLCA.

Each host language provides a mechanism for handling diagnostic information.

- In Assembler, C, COBOL, Fortran, and PL/I, an application program that contains executable SQL statements must provide at least one of the following:
  - A structure named SQLCA, which can be provided by using the INCLUDE SQLCA statement
  - A stand-alone CHAR(5) (CHAR(6) in C) variable named *SQLSTATE* (*SQLSTT* in Fortran)
  - A stand-alone integer variable named *SQLCODE* (*SQLCOD* in Fortran)
- In Java, for error conditions, the getSQLState method of the JDBC SQLException class can be used to get the SQLSTATE and the getErrorCode method can be used to get the SQLCODE.
- In REXX, an SQLCA is provided automatically.

Whether you define stand-alone SQLCODE and SQLSTATE host variables or an SQLCA in your program depends on the DB2 precompiler option you choose.

If the application is using DB2 ODBC and it calls the SQLGetSQLCA function, it need only include an SQLCA. Otherwise, all notification of success or errors is specified with return codes for the various function calls.

When you specify STDSQL(YES), which indicates conformance to the SQL standard, you should not define an SQLCA. The stand-alone variable for SQLCODE must be a valid host variable in the DECLARE SECTION of a program.

It can also be declared outside of the DECLARE SECTION when no variable is defined for SQLSTATE. The stand-alone variable for SQLSTATE must be declared in the DECLARE SECTION. It must not be declared as an element of a structure.

Use a stand-alone SQLSTATE to conform with the SQL 2003 Core standard. When you specify STDSQL(NO), which indicates conformance to DB2 rules, you must include an SQLCA explicitly to have access to the SQLSTATE and SQLCODE information.

SQLSTATE

DB2 sets SQLSTATE after each SQL statement (other than GET DIAGNOSTICS or a compound statement) is executed. DB2 returns values that conform to the error specification in the SQL standard. Thus, application programs can check the execution of SQL statements by testing SQLSTATE instead of SQLCODE.

SQLSTATE provides application programs with common codes for common error conditions (the values of SQLSTATE are product-specific if the error or warning is product-specific). Furthermore, SQLSTATE is designed so that application programs can test for specific errors or classes of errors. The coding scheme is the same for all IBM implementations of SQL. The SQLSTATE values are based on the SQLSTATE specifications contained in the SQL standard. Error messages and the tokens that are substituted for variables in error messages are associated with SQLCODE values, not SQLSTATE values.

In the case of a LOOP statement, the SQLSTATE is set after the END LOOP portion of the LOOP statement completes. With the REPEAT statement, the SQLSTATE is set after the UNTIL and END REPEAT portions of the REPEAT statement completes.

If the application is using DB2 ODBC, the SQLSTATE returned conforms to the ODBC Version 2.0 specification.

SQLCODE

The SQLCODE is also set by DB2 after each SQL statement is executed as follows:

DB2 conforms to the SQL standard as follows:

- If SQLCODE = 0 and SQLWARN0 is blank, execution was successful.
- If SQLCODE = 100, "no data" was found. For example, a FETCH statement returned no data because the cursor was positioned after the last row of the result table.
- If SQLCODE > 0 and not = 100, execution was successful with a warning.
- If SQLCODE = 0 and SQLWARN0 = 'W', execution was successful with a warning.
- If SQLCODE < 0, execution was not successful.

In the case of a LOOP statement, the SQLSTATE is set after the END LOOP portion of the LOOP statement completes. With the REPEAT statement, the SQLSTATE is set after the UNTIL and END REPEAT portions of the REPEAT statement completes.

The SQL standard does not define the meaning of any other specific positive or negative values of SQLCODE, and the meaning of these values is not the same in all implementations of SQL.

If the application is using DB2 ODBC, an SQLCODE is only returned if the application issues the SQLGetSQLCA function.

SQL comments

Static SQL statements can include host language or SQL comments. Dynamic SQL statements can include SQL comments. There are two types of SQL comments, simple comments and bracketed comments.

simple comments

Simple comments are introduced with two consecutive hyphens (--) and end with the end of a line. The following rules apply to the use of simple comments:

- The two hyphens must be on the same line and must not be separated by a space.
- Simple comments can be started whenever a space is valid (except within a delimiter token or between 'EXEC' and 'SQL').
- Simple comments cannot be continued to the next line.
- In COBOL, the hyphen must be preceded by a space.

bracketed comments

Bracketed comments are introduced with /* and end with */. The following rules apply to the use of bracketed comments:

- The /* must be on the same line and not separated by a space.
- The */ must be one the same line and not separated by a space.
- Bracketed comments can be started wherever a space is valid (except within a delimiter token or between 'EXEC' and 'SQL').
- Bracketed comments can be continued to the next line.
- Bracketed comments can be nested within other bracketed comments. However, nested bracketed comments are not supported by DSNTEP2, DSNTEP4, SPUFI, or the command line processor.
- Bracketed comments are not allowed in static SQL statements in a COBOL, Fortran, or Assembler program.

Example: This example shows how to include comments in an SQL statement within a C program. The example uses both simple and bracketed comments:

```
EXEC SQL
  CREATE VIEW PRJ_MAXPER         --projects with most support personnel
    /*
     * Returns number and name of the project
     */
   AS SELECT PROJNO, PROJNAME     -- number and name of project
        FROM DSN8910.PROJ
     /*
      * E21 is the systems support dept code
      */
   WHERE DEPTNO = 'E21'           -- systems support dept code
   AND PRSTAFF > 1;
```

For information about host language comments, refer to *DB2 Application Programming and SQL Guide*.

ALLOCATE CURSOR

The ALLOCATE CURSOR statement defines a cursor and associates it with a result set locator variable.

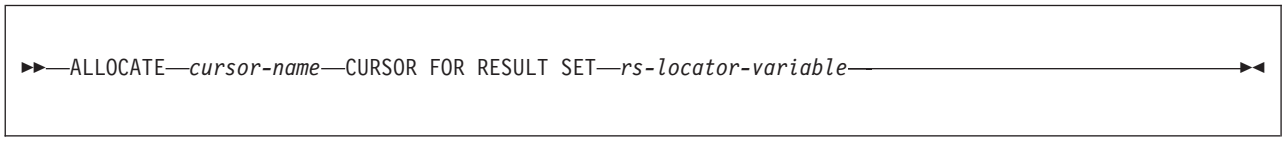
Invocation

This statement can be embedded in an application program. It is an executable statement that can be dynamically prepared. It cannot be issued interactively.

Authorization

None required.

Syntax



Description

cursor-name
Names the cursor. The name must not identify a cursor that has already been declared in the source program.

CURSOR FOR RESULT SET *rs-locator-variable*
Specifies a result set locator variable that has been declared in the application program according to the rules for declaring result set locator variables.

The result set locator variable must contain a valid result set locator value, as returned by the ASSOCIATE LOCATORS or DESCRIBE PROCEDURE SQL statement. The value of the result set locator variable is used at the time the cursor is allocated. Subsequent changes to the value of the result set locator have no affect on the allocated cursor. The result set locator value must not be the same as a value used for another cursor allocated in the source program.

Notes

Dynamically prepared ALLOCATE CURSOR statements: The EXECUTE statement with the USING clause must be used to execute a dynamically prepared ALLOCATE CURSOR statement. In a dynamically prepared statement, references to host variables are represented by parameter markers (question marks). In the ALLOCATE CURSOR statement, *rs-locator-variable* is always a host variable. Thus, for a dynamically prepared ALLOCATE CURSOR statement, the USING clause of the EXECUTE statement must identify the host variable whose value is to be substituted for the parameter marker that represents *rs-locator-variable*.

You cannot prepare an ALLOCATE CURSOR statement with a statement identifier that has already been used in a DECLARE CURSOR statement. For example, the following SQL statements are invalid because the PREPARE statement uses STMT1 as an identifier for the ALLOCATE CURSOR statement and STMT1 has already been used for a DECLARE CURSOR statement.

```
DECLARE CURSOR C1 FOR STMT1;
PREPARE STMT1 FROM          INVALID
  'ALLOCATE C2 CURSOR FOR RESULT SET ?';
```

Rules for using an allocated cursor: The following rules apply when you use an allocated cursor:

- You cannot open an allocated cursor with the OPEN statement.
- You can close an allocated cursor with the CLOSE statement. Closing an allocated cursor closes the associated cursor defined in the stored procedure.
- You can allocate only one cursor to each result set.

The life of an allocated cursor: A rollback operation, an implicit close, or an explicit close destroy allocated cursors. A commit operation destroys allocated cursors that are not defined WITH HOLD by the stored procedure. Destroying an allocated cursor closes the associated cursor defined in the stored procedure.

Considerations for scrollable cursors: Following an ALLOCATE CURSOR statement, a GET DIAGNOSTICS statement can be used to get the attributes of the cursor such as the following information (for more information, see "GET DIAGNOSTICS" on page 1679):

- DB2_SQL_ATTR_CURSOR_HOLD. Whether the cursor was defined with the WITH HOLD attribute.
- DB2_SQL_ATTR_CURSOR_SCROLLABLE. Scrollability of the cursor.
- DB2_SQL_ATTR_CURSOR_SENSITIVITY. Effective sensitivity of the cursor.

  The sensitivity information can be used by applications (such as an ODBC driver) to determine what type of FETCH (INSENSITIVE or SENSITIVE) to issue for a cursor defined as ASENSITIVE.
- DB2_SQL_ATTR_CURSOR_ROWSET. Whether the cursor can be used to access rowsets.
- DB2_SQL_ATTR_CURSOR_TYPE. Whether a cursor type is forward-only, static, or dynamic.
- The scrollability of the cursor is in SQLWARN1.
- The sensitivity of the cursor is in SQLWARN4.
- The effective capability of the cursor is in SQLWARN5.

Example

The statement in the following example is assumed to be in a PL/I program.

Define and associate cursor C1 with the result set locator variable *LOC1* and the related result set returned by the stored procedure:

```
EXEC SQL ALLOCATE C1 CURSOR FOR RESULT SET :LOC1;
```

ALTER DATABASE

The ALTER DATABASE statement changes the description of a database at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

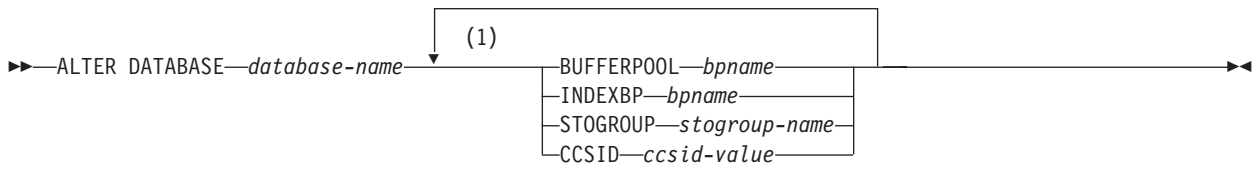
Authorization

The privilege set that is defined below must include at least one of the following:
- The DROP privilege on the database
- Ownership of the database
- DBADM or DBCTRL authority for the database
- SYSADM or SYSCTRL authority
- System DBADM

If the database is implicitly created, the privileges must be on the implicit database or on DSNDB04.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the package. If the statement is dynamically prepared, the privilege set is the union of the privilege sets that are held by each authorization ID and role of the process.

Syntax



Notes:

1 The same clause must not be specified more than one time.

Description

DATABASE *database-name*

Identifies the database that is to be altered. The name must identify a database that exists at the current server and must not identify an implicitly created system database.

BUFFERPOOL *bpname*

Identifies the default buffer pool for the table spaces within the database. It does not apply to table spaces that already exist within the database.

If the database is a work file database, 8 KB and 16 KB buffer pools cannot be specified.

See “Naming conventions” on page 57 for more details about *bpname*.

INDEXBP *bpname*
Identifies the default buffer pool for the indexes within the database. It does not apply to indexes that already exist within the database. The name can identify a 4 KB, 8 KB, 16 KB, or 32 KB buffer pool. See “Naming conventions” on page 57 for more details about *bpname*.

STOGROUP *stogroup-name*
Identifies the storage group to be used, as required, as a default storage group to support DASD space requirements for table spaces and indexes within the database. It does not apply to table spaces and indexes that already exist within the database.

CCSID *ccsid-value*
Identifies the default CCSID for table spaces within the database. It does not apply to existing table spaces in the database. *ccsid-value* must identify a CCSID value that is compatible with the current value of the CCSID for the database. “Notes” contains a list that shows the CCSID to which a given CCSID can be altered.

CCSID cannot be specified for a work file database.

Notes

Altering the CCSID: The ability to alter the default CCSID enables you to change to a CCSID that supports the Euro symbol. You can only convert between specific CCSIDs that do and do not define the Euro symbol. In most cases, the code point that supports the Euro symbol replaces an existing code point, such as the International Currency Symbol (ICS).

Changing a CCSID can be disruptive to the system and requires several steps. For each encoding scheme of a system (ASCII, EBCDIC, and Unicode), DB2 supports SBCS, DBCS, and mixed CCSIDs. Therefore, the CCSIDs for all databases and all table spaces within an encoding scheme should be altered at the same time. Otherwise, unpredictable results might occur.

The recommended method for changing the CCSID requires that the data be unloaded and reloaded. See *DB2 Installation Guide* for the steps needed to change the CCSID, such as running an installation CLIST to modify the CCSID data in DSNHDECP, when to drop and re-create views, and when to rebind invalidated packages.

The following lists show the CCSIDs that can be converted. The second CCSID in each pair is the CCSID with the Euro symbol. The CCSID can be changed from the CCSID that does not support the Euro symbol to the CCSID that does, and vice versa. For example, if the current CCSID is 500, it can be changed to 1148.

```
EBCDIC CCSIDs
---------------

37         1140
273        1141
277        1142
278        1143
280        1144
284        1145
285        1146
297        1147
500        1148
871        1149
```

```
ASCII CCSIDs
---------------

850        858
874        4970
1250       5346
1251       5347
1252       5348
1253       5349
1254       5350
1255       5351
1256       5352
1257       5353
```

Example

Change the default buffer pool for both table spaces and indexes within database ABCDE to BP2.

```
ALTER DATABASE ABCDE
  BUFFERPOOL BP2
  INDEXBP BP2;
```

ALTER FUNCTION (external)

The ALTER FUNCTION statement changes the description of a user-defined external scalar function or external table function at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set defined below must include at least one of the following:

- Ownership of the function
- The ALTERIN privilege on the schema
- SYSADM or SYSCTRL authority
- System DBADM

The authorization ID that matches the schema name implicitly has the ALTERIN privilege on the schema.

If the authorization ID that is used to alter the function has installation SYSADM authority, the function is identified as system-defined function when the function definition is reevaluated.

At least one of the following privileges is required if the SECURED option is specified or if the function is currently secured and the NOT SECURED option is specified:

- SECADM authority
- CREATE_SECURE_OBJECT privilege

For *external scalar functions*, when **LANGUAGE** is **JAVA** and a *jar-name* is specified in the **EXTERNAL NAME** clause, the privilege set must include USAGE on the JAR file.

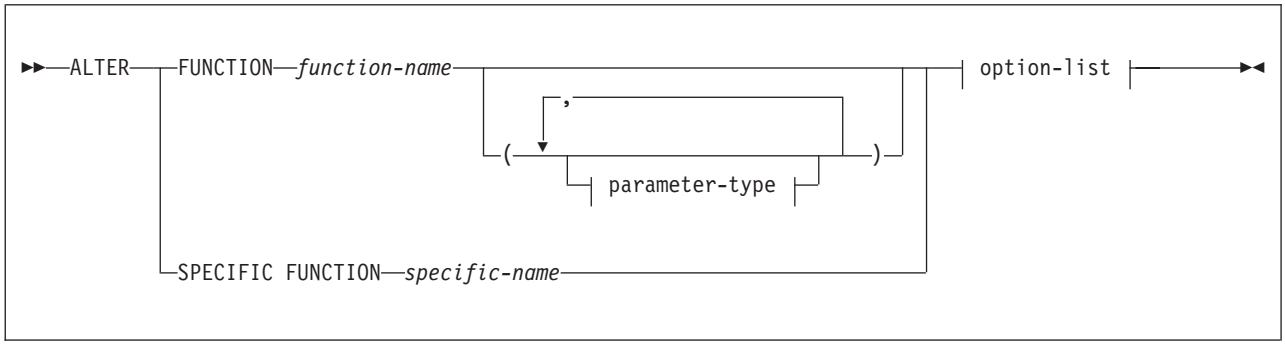
Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the package.

If the statement is dynamically prepared, the privilege set is the set of privileges that are held by the SQL authorization IDs of the process. The specified routine name can include a schema name (a qualifier). However, if the schema name is not the same as one of these SQL authorization IDs, one of the following conditions must be met:

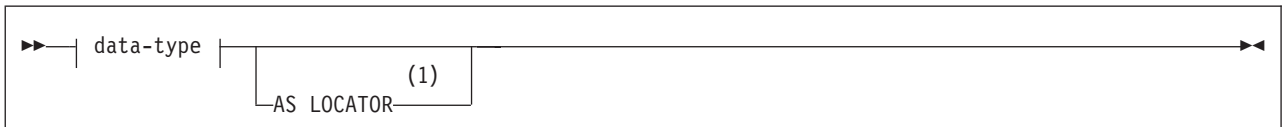
- The privilege set includes SYSADM authority
- The privilege set includes SYSCTRL authority
- The SQL authorization ID of the process has the ALTERIN privilege on the schema

If the environment in which the function is to be run is being changed, the authorization ID must have authority to use the WLM environment specified. The required authorization is obtained from an external security product, such as RACF.

Syntax



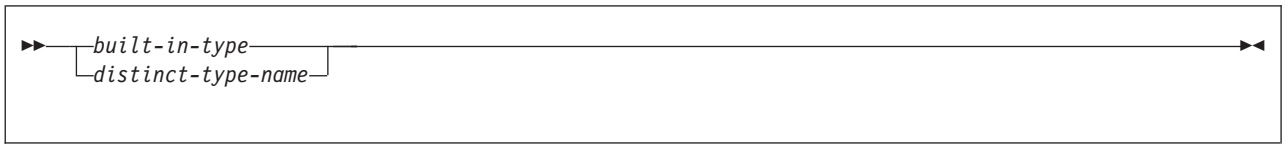
parameter-type:



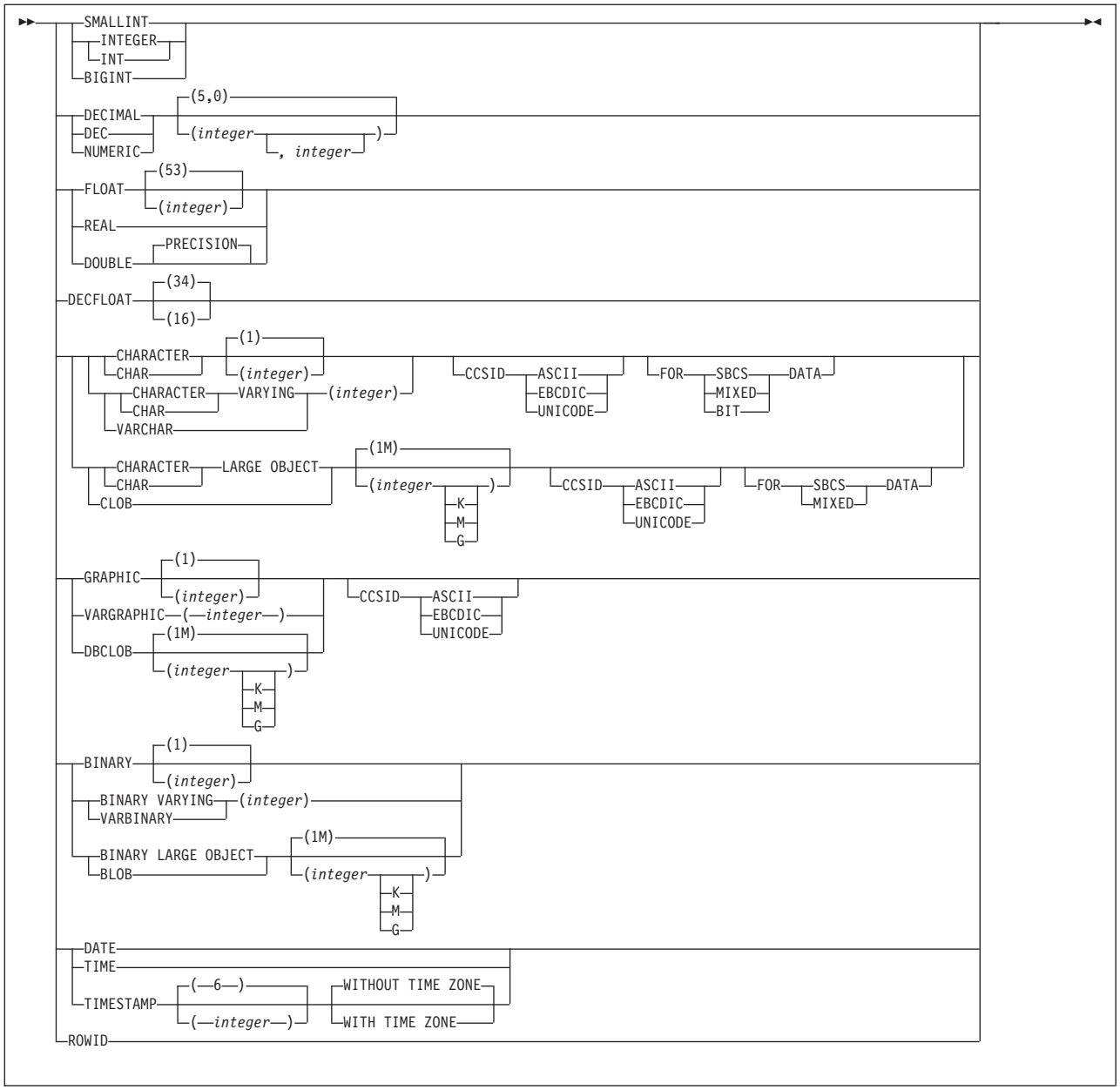
Notes:

1 **AS LOCATOR** can be specified only for a LOB data type or a distinct type based on a LOB data type.

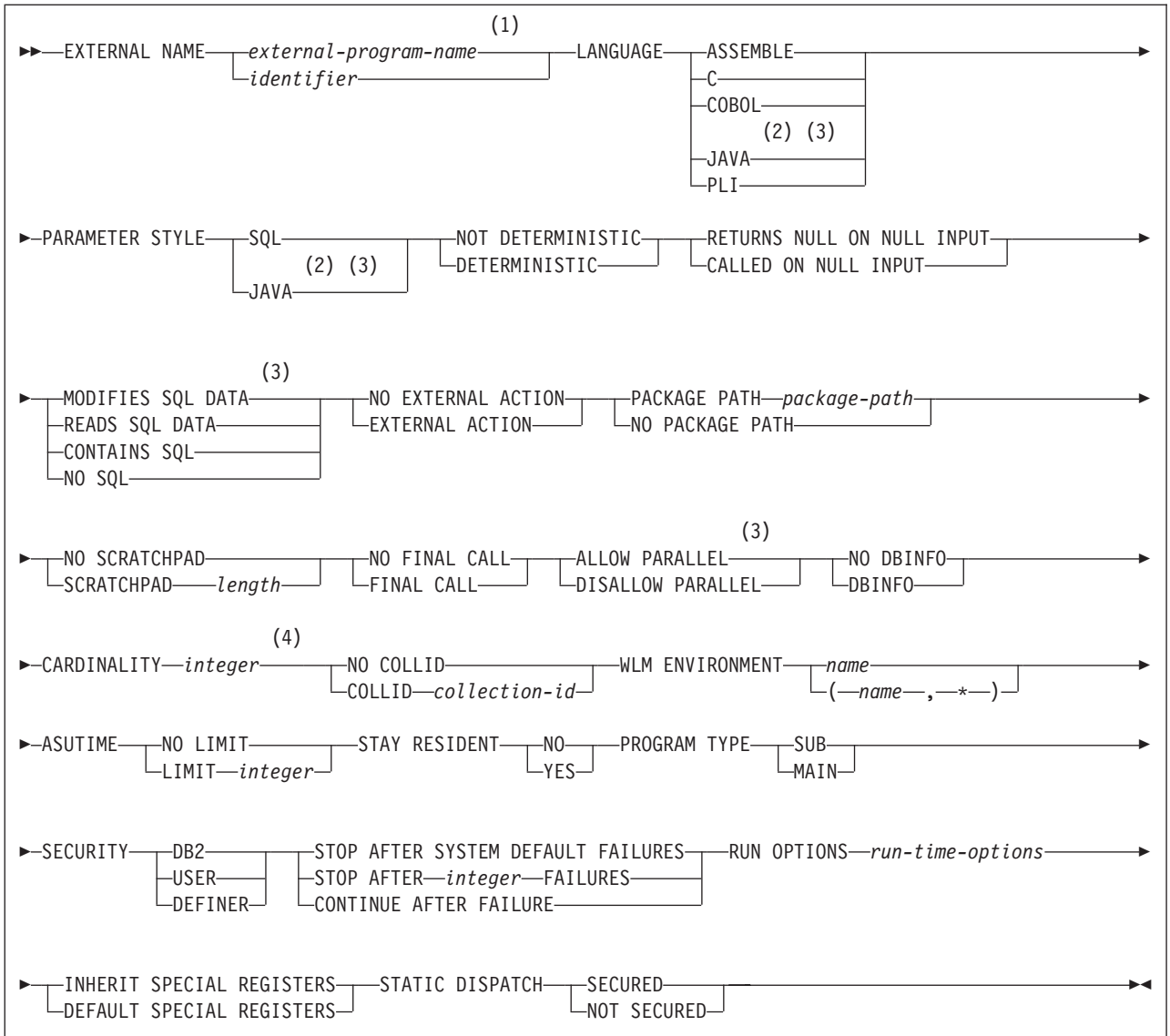
data-type:



built-in-type:



option-list: (Specify options in any order. Specify at least one option. Do not specify the same option more than once.)



Notes:

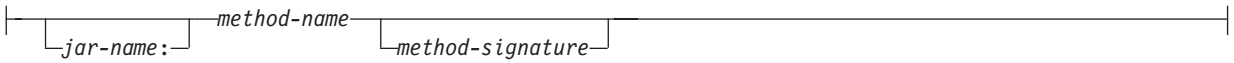
1 If **LANGUAGE** is **JAVA**, **EXTERNAL NAME** must be specified with a valid *external-java-routine-name*.

2 When **LANGUAGE JAVA** is specified, **PARAMETER STYLE JAVA** must also be specified. When **PARAMETER STYLE JAVA** is specified, **LANGUAGE JAVA** must also be specified.

3 **LANGUAGE JAVA**, **PARAMETER STYLE JAVA**, **MODIFIES SQL DATA**, and **ALLOW PARALLEL** are not supported for *external table functions*.

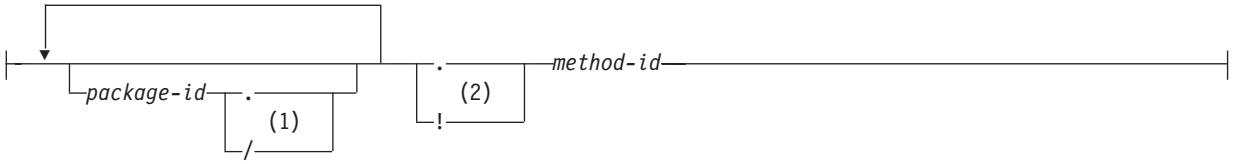
4 **CARDINALITY** is not supported for *external scalar functions*.

external-java-routine-name:



jar-name:

method-name:



method-signature:

Notes:

1 The slash (/) is supported for compatibility with previous releases of DB2 for z/OS.

2 The exclamation point (!) is supported for compatibility with other products in the DB2 family.

Description

One of the following three clauses identifies the function to be changed.

FUNCTION *function-name*
Identifies the external function by its function name. *function-name* must identify a function that exists at the current server. The function must be a user-defined external function, and there must be exactly one function with *function-name* in the schema.

The function can have any number of input parameters. If the schema does not contain a function with *function-name* or contains more than one function with this name, an error occurs.

FUNCTION *function-name (parameter-type,...)*
Identifies the external function by its function signature, which uniquely identifies the function.

function-name
Identifies the function by its name.

If *function-name()* is specified, the function that is identified must have zero parameters.

(*parameter-type***,...)**
Identifies the number of input parameters of the function and their data types.

The data type of each parameter must match the data type that was specified in the CREATE FUNCTION statement for the parameter in the corresponding position. The number of data types and the logical concatenation of the data types are used to uniquely identify the function. Therefore, you cannot change the number of parameters or the data types of the parameters.

For data types that have a length, precision, or scale attribute, you can use a set of empty parentheses, specify a value, or accept the default values:

If the function was defined with a table parameter (the **LIKE TABLE** *name* **AS LOCATOR** clause was specified in the CREATE FUNCTION statement to indicate that one of the input parameters is a transition table), the function signature cannot be used to uniquely identify the function. Instead, use one of the other syntax variations to identify the function with its function name, if unique, or its specific name.

- Empty parentheses indicate that DB2 is to ignore the attribute when determining whether the data types match.

  For example, DEC() will be considered a match for a parameter of a function defined with a data type of DEC(7,2). Similarly DECFLOAT() will be considered a match for DECFLOAT(16) or DECFLOAT(34).

  FLOAT cannot be specified with empty parentheses because its parameter value indicates different data types (REAL or DOUBLE).
- If you use a specific value for a length, precision, or scale attribute, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

  The specific value for FLOAT(*n*) does not have to exactly match the defined value of the source function because `1<=`*n*`<=21` indicates REAL and `22<=`*n*`<=53` indicates DOUBLE. Matching is based on whether the data type is REAL or DOUBLE.
- If length, precision, or scale is not explicitly specified and empty parentheses are not specified, the default length of the data type is implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

For data types with a subtype or encoding scheme attribute, specifying the **FOR** *subtype* **DATA** clause or the **CCSID** clause is optional. Omission of either clause indicates that DB2 is to ignore the attribute when determining whether the data types match. If you specify either clause, it must match the value that was implicitly or explicitly specified in the CREATE FUNCTION statement.

See “CREATE FUNCTION” on page 1165 for more information on the specification of the parameter list.

A function with the function signature must exist in the explicitly or implicitly specified schema.

SPECIFIC FUNCTION *specific-name*
Identifies the external function by its specific name. A function with the specific name must exist in the schema.

The following clauses change the description of the function that has been identified to be changed.

EXTERNAL NAME *external-program-name* **or** *identifier*
Identifies the user-written code (program) that runs when the function is invoked.

If **LANGUAGE** is **JAVA**, *external-program-name* must be specified and enclosed in single quotation marks, with no extraneous blanks within the single quotation marks. It must specify a valid *external-java-routine-name*. If multiple *external-program-name* values are specified, the total length of all of them must not be greater than 1305 bytes and they must be separated by a space or a line break. Do not specify a JAR file for a Java function for which **NO SQL** is in effect.

An *external-java-routine-name* contains the following parts:

jar-name
Identifies the name given to the JAR file when it was installed in the database. The name contains *jar-id*, which can optionally be qualified with a schema. Examples are "myJar" and "mySchema.myJar." The unqualified *jar-id* is implicitly qualified with a schema name according to the following rules:

- If the statement is embedded in a program, the schema name is the authorization ID in the QUALIFIER option of the **BIND** subcommand for a package or plan when the package or plan was created or last changed. The schema name can also be the authorization ID in the QUALIFIER option of the CREATE PROCEDURE or ALTER PROCEDURE statement for a native SQL procedure when the procedure was created or last changed. If the QUALIFIER is not specified, the schema name is the owner of the package, plan, or native SQL procedure.
- If the statement is dynamically prepared, the schema name is the SQL authorization ID in the CURRENT SCHEMA special register.

If *jar-name* is specified, it must exist when the ALTER FUNCTION statement is processed.

If *jar-name* is not specified, the function is loaded from the class file directly. DB2 searches the directories in the CLASSPATH associated with the WLM Environment. Environmental variables for Java routines are specified in a data set identified in a JAVAENV DD card on the JCL used to start the address space for a WLM-managed function.

method-name
Identifies the name of the method and must not be longer than 254 bytes. Its package, class, and method IDs are specific to Java and as such are not limited to 18 bytes. In addition, the rules for what method IDs can contain are not necessarily the same as the rules for an SQL ordinary identifier.

package-id
Identifies a package. The concatenated list of *package-ids* identifies the package that the class identifier is part of. If the class is part of a package, the method name must include the complete package prefix, such as "myPacks.UserFuncs." The Java virtual machine looks in the directory "/myPacks/UserFuncs/" for the classes.

class-id
Identifies the class identifier of the Java object.

method-id
Identifies the method identifier with the Java class to be invoked.

method-signature
Identifies a list of zero or more Java data types for the parameter list and must not be longer than 1024 bytes. Specify the *method-signature* if the user-defined function involves any input or output parameters that can be NULL. When the function that is being created is called, DB2 searches for a Java method with the exact *method-signature*. The number of *java-datatype* elements that are specified indicates how many parameters that the Java method must have.

A Java procedure can have no parameters. In this case, you code an empty set of parentheses for *method-signature*. If a Java *method-signature* is not specified, DB2 searches for a Java method with a signature derived from the default JDBC types associated with the SQL types specified in the parameter list of the ALTER FUNCTION statement.

For other values of **LANGUAGE**, the value must conform to the naming conventions for load modules: the value must be less than or equal to 8 bytes, and it must conform to the rules for an ordinary identifier with the exception that it must not contain an underscore.

LANGUAGE
Specifies the application programming language in which the function is written. All programs must be designed to run in IBM's Language Environment environment.

ASSEMBLE
The function is written in Assembler.

C The function is written in C or C++.

COBOL
The function is written in COBOL, including the object-oriented language extensions.

JAVA
The user-defined function is written in Java and is executed in the Java virtual machine. If the ALTER FUNCTION statement results in changing **LANGUAGE** to **JAVA**, **PARAMETER STYLE JAVA** and an **EXTERNAL NAME** clause must be specified to provide the appropriate values. When **LANGUAGE JAVA** is specified, the **EXTERNAL NAME** clause must also be specified with a valid *external-java-routine-name* and **PARAMETER STYLE** must be specified with **JAVA**.

Do not specify **LANGUAGE JAVA** when **SCRATCHPAD**, **FINAL CALL**, **DBINFO**, **PROGRAM TYPE MAIN**, or **RUN OPTIONS** is specified. Do not specify **LANGUAGE JAVA** for a table function.

PLI
The function is written in PL/I.

PARAMETER STYLE
Specifies the linkage convention that the function program uses to receive input parameters from and pass return values to the invoking SQL statement.

SQL
Specifies the parameter passing convention that supports passing null values both as input and for output. The parameters that are passed between the invoking SQL statement and the function include:

- Input parameters. The first *n* parameters are the input parameters that are specified for the function.
- Result parameters. For an external scalar function, a parameter for the result of the function. For an external table function, the next *m* parameters that are specified on the RETURNS TABLE clause of the CREATE statement that defined the function.
- Input parameter indicator variables. *n* parameters for the indicator variables for the input parameters.
- Result parameter indicator variables. For an external scalar function, a parameter for the indicator variable for the result of the function that is specified on the RETURNS clause of the CREATE statement that defined the function. For an external table function, *m* parameters for the indicator variables of the result columns of the function that are specified on the RETURNS TABLE clause of the CREATE statement that defined the function.
- The SQLSTATE to be returned to DB2.
- The qualified name of the function.
- The specific name of the function.
- The SQL diagnostic string to be returned to DB2.
- The scratchpad, if **SCRATCHPAD** is specified.
- The call type. For an external scalar function, the call type is passed only if **FINAL CALL** is specified. The call type is always passed for an external table function.
- The DBINFO structure, if **DBINFO** is specified.

JAVA
Indicates that the user-defined function uses a convention for passing parameters that conforms to the Java and SQLJ specifications. If the ALTER FUNCTION statement results in changing **LANGUAGE** to **JAVA**, **PARAMETER STYLE JAVA** and an **EXTERNAL NAME** clause must be specified to provide the appropriate values. **PARAMETER STYLE JAVA** can be specified only if **LANGUAGE** is **JAVA**. **JAVA** must be specified for **PARAMETER STYLE** when **LANGUAGE** is **JAVA**.

Do not specify **PARAMETER STYLE JAVA** for a table function.

NOT DETERMINISTIC or DETERMINISTIC
Specifies whether the function returns the same results each time that the function is invoked with the same input arguments.

NOT DETERMINISTIC
The function might not return the same result each time that the function is invoked with the same input arguments. The function depends on some state values that affect the results. DB2 uses this information to disable the merging of views and table expressions when processing SELECT or SQL data change statements that refer to this function. An example of a function that is not deterministic is one that generates random numbers, or any function that contains SQL statements.

Some SQL functions that invoke functions that are not deterministic can receive incorrect results if the function is executed by parallel tasks. Specify the **DISALLOW PARALLEL** clause for these functions.

If a view or a materialized query table definition refers to the function, the function cannot be changed to **NOT DETERMINISTIC**. To change the function, drop any views or materialized query tables that refer to the function first.

DETERMINISTIC
The function always returns the same result each time that the function is invoked with the same input arguments. An example of a deterministic function is a function that calculates the square root of the input. DB2 uses this information to enable the merging of views and table expressions for SELECT or SQL data change statements that refer to this function. If applicable, specify DETERMINISTIC to prevent non-optimal access paths from being chosen for SQL statements that refer to this function.

DB2 does not verify that the function program is consistent with the specification of **DETERMINISTIC** or **NOT DETERMINISTIC**.

RETURNS NULL ON NULL INPUT or CALLED ON NULL INPUT
Specifies whether the function is called if any of the input arguments is null at execution time.

RETURNS NULL ON NULL INPUT
The function is not called if any of the input arguments is null. For an external scalar function, the result is the null value. For an external table function, the result is an empty table, which is a table with no rows.

CALLED ON NULL INPUT
The function is called regardless of whether any of the input arguments are null, making the function responsible for testing for null argument values. For an external scalar function, the function can return a null or nonnull value. For an external table function, the function can return an empty table, depending on its logic.

MODIFIES SQL DATA, READS SQL DATA, CONTAINS SQL, or NO SQL
Specifies which SQL statements, if any, can be executed in the function or any routine that is called from this function.

MODIFIES SQL DATA
Specifies that the function can execute any SQL statement except the statements that are not supported in functions. Do not specify **MODIFIES SQL DATA** when **ALLOW PARALLEL** is in effect.

READS SQL DATA
Specifies that the function can execute statements with a data access indication of READS SQL DATA, CONTAINS SQL, or NO SQL. The function cannot execute SQL statements that modify data.

CONTAINS SQL
Specifies that the function can execute only SQL statements with a data classification of CONTAINS SQL or NO SQL. SQL statements that neither read nor modify SQL data can be executed by the function. Statements that are not supported in any function return a different error.

NO SQL
Specifies that the function can execute only SQL statements with a data access classification of NO SQL. Do not specify **NO SQL** for a Java function that uses a JAR file.

NO EXTERNAL ACTION or EXTERNAL ACTION
Specifies whether the function takes an action that changes the state of an object that DB2 does not manage. An example of an external action is sending a message or writing a record to a file.

Because DB2 uses the RRS attachment for external functions, DB2 can participate in two-phase commit with any other resource manager that uses RRS. For resource managers that do not use RRS, there is no coordination of commit or rollback operations on non-DB2 resources.

NO EXTERNAL ACTION
The function does not take any action that changes the state of an object that DB2 does not manage. DB2 uses this information to enable the merging of views and table expressions for SELECT or SQL data change statements that refer to this function. If applicable, specify **NO EXTERNAL ACTION** to prevent non-optimal access paths from being chosen for SQL statements that refer to this function.

EXTERNAL ACTION
The function can take an action that changes the state of an object that DB2 does not manage.

Some SQL statements that invoke functions with external actions can result in incorrect results if parallel tasks execute the function. For example, if the function sends a note for each initial call to it, one note is sent for each parallel task instead of once for the function. Specify the **DISALLOW PARALLEL** clause for functions that do not work correctly with parallelism.

If you specify **EXTERNAL ACTION**, DB2:

- Materializes the views and table expressions in SELECT or SQL data change statements that refer to the function. This materialization can adversely affect the access paths that are chosen for the SQL statements that refer to this function. Do not specify EXTERNAL ACTION if the function does not have an external action.
- Does not move the function from one task control block (TCB) to another between FETCH operations.
- Does not allow another function or stored procedure to use the TCB until the cursor is closed. This is also applicable for cursors declared WITH HOLD.

The only changes to resources made outside of DB2 that are under the control of commit and rollback operations are those changes made under RRS control.

If a view or a materialized query table definition refers to the function, the function cannot be changed to **EXTERNAL ACTION**. To change the function, drop any views or materialized query tables that refer to the function first.

DB2 does not verify that the function program is consistent with the specification of **EXTERNAL ACTION** or **NO EXTERNAL ACTION**.

NO PACKAGE PATH or PACKAGE PATH *package-path*
Identifies the package path to use when the function is run. This is the list of the possible package collections into which the DBRM that is associated with the function is bound.

NO PACKAGE PATH
Specifies that the list of package collections for the function is the same as

the list of package collections for the program that invokes the function. If the program that invokes the function does not use a package, DB2 resolves the package by using the CURRENT PACKAGE PATH special register, the CURRENT PACKAGESET special register, or the PKLIST bind option (in this order). For information about how DB2 uses these three items, see *DB2 Application Programming and SQL Guide*.

PACKAGE PATH *package-path*
Specifies a list of package collections, in the same format as used in the SET CURRENT PACKAGE PATH statement.

If the **COLLID** clause is specified with **PACKAGE PATH**, the **COLLID** clause is ignored when the function is invoked.

The *package-path* value that is associated with the function definition is checked when the function is invoked. If *package-path* contains SESSION_USER (or USER), PATH, or PACKAGE PATH, an error is returned when the *package-path* value is checked.

NO SCRATCHPAD or SCRATCHPAD
Specifies whether DB2 is to provide a scratchpad for the function. Using reentrant external functions and a scratchpad (which provides an area for the function to save information from one invocation to the next) is strongly recommended.

NO SCRATCHPAD
A scratchpad is not allocated and passed to the function.

SCRATCHPAD *length*
When the function is invoked for the first time, DB2 allocates memory for a scratchpad. A scratchpad has the following characteristics:

- *length* must be between 1 and 32767. The default value is 100 bytes.
- DB2 initializes the scratchpad to all binary zeros (X'00').
- The scope of a scratchpad is the SQL statement. For each reference to the function in an SQL statement, there is one scratchpad.

  For example, assuming that user-defined function UDFX is a scalar function that is defined with the **SCRATCHPAD** option, three scratchpads are allocated for the three references to UDFX in the following SQL statement:

  ```
  SELECT A, UDFX(A) FROM TABLEB
    WHERE UDFX(A) > 103 OR UDFX(A) < 19;
  ```

  For another example, assume that UDFX is a user-defined table function that is defined with the **SCRATCHPAD** option. Two scratchpads are allocated for the two references to function UDFX in the following SQL statement:

  ```
  SELECT *
    FROM TABLE (UDFX(A)), TABLE (UDFX(B));
  ```

  If the function is run under parallel tasks, one scratchpad is allocated for each parallel task of each reference to the function in the SQL statement. This can lead to unpredictable results. For example, if a function uses the scratchpad to count the number of times that it is invoked, the count reflects the number of invocations done by the parallel task and not the SQL statement. Specify the **DISALLOW PARALLEL** clause for functions that do not work correctly with parallelism.
- The scratchpad is persistent. DB2 preserves its content from one invocation of the function to the next. Any changes that the function makes to the scratchpad on one call are still there on the next call. DB2

initializes the scratchpads when it begins to execute an SQL statement. DB2 does not reset scratchpads when a correlated subquery begins to execute.

- The scratchpad can be a central point for the system resources that the function acquires. If the function acquires system resources, specify **FINAL CALL** to ensure that DB2 calls the function one more time so that the function can free those system resources.

Each time that the function is invoked, DB2 passes an additional argument to the function that contains the address of the scratchpad.

If you specify **SCRATCHPAD**, DB2:

- Does not move the function from one TCB or address space to another between FETCH operations.
- Does not allow another function or stored procedure to use the TCB until the cursor is closed. This is also applicable for cursors declared WITH HOLD.

Do not specify **SCRATCHPAD** when **LANGUAGE JAVA** is specified.

NO FINAL CALL or FINAL CALL
Specifies whether a final call is made to the function. A *final call* enables the function to free any system resources that it has acquired. A final call is useful when the function has been defined with the **SCRATCHPAD** keyword and the function acquires system resource and anchors them in the scratchpad.

The effect of **NO FINAL CALL** or **FINAL CALL** depends on whether the external function is a scalar function or a table function.

For an external scalar function:

NO FINAL CALL
A final call is not made to the external scalar function. The function does not receive an additional argument that specifies the type of call.

FINAL CALL
A final call is made to the external scalar function. See the following description of call types for the characteristics of a final call. When **FINAL CALL** is specified, the function receives an additional argument that specifies the type of call to enable the function to differentiate between a final call and another type of call. Do not specify **FINAL CALL** when **LANGUAGE JAVA** is specified.

For more information on **NO FINAL CALL** and **FINAL CALL** for external scalar functions, including the types of calls, see the description of the option for "CREATE FUNCTION (external scalar)" on page 1166.

For an external table function:

NO FINAL CALL
A first and final call are not made to the external table function.

FINAL CALL
A first call and final call are made to the external table function in addition to one or more other types of calls.

For both **NO FINAL CALL** and **FINAL CALL**, the function receives an additional argument that specifies the type of call. For more information on

NO FINAL CALL and FINAL CALL for external table functions, including the types of calls, see the description of the option for "CREATE FUNCTION (external table)" on page 1191.

ALLOW or DISALLOW PARALLEL
Specifies whether, for a single reference to the function, the function can be executed in parallel. If the function is defined with **MODIFIES SQL DATA**, specify **DISALLOW PARALLEL**, not **ALLOW PARALLEL**.

ALLOW PARALLEL
Specifies that DB2 can consider parallelism for the function. Parallelism is not forced on the SQL statement that invokes the function or on any SQL statement in the function. Existing restrictions on parallelism apply.

See **SCRATCHPAD**, **EXTERNAL ACTION**, and **FINAL CALL** for considerations when specifying **ALLOW PARALLEL**.

DISALLOW PARALLEL
Specifies that DB2 does not consider parallelism for the function.

NO DBINFO or DBINFO
Specifies whether additional status information is passed to the function when it is invoked.

NO DBINFO
Additional information is not passed.

DBINFO
An additional argument is passed when the function is invoked. The argument is a structure that contains information such as the application run time authorization ID, the schema name, the name of a table or column that the function might be inserting into or updating, and identification of the database server that invoked the function. For details about the argument and its structure, see *DB2 Application Programming and SQL Guide*.

Do not specify **DBINFO** when **LANGUAGE JAVA** is specified.

CARDINALITY *integer*
Specifies an estimate of the expected number of rows that the function returns. The number is used for optimization purposes. The value of *integer* must range from 0 to 2147483647.

If a function has an infinite cardinality (which means that the function never returns the "end-of-table" condition and always returns a row), a query that requires the end-of-table condition to work correctly needs to be interrupted. Thus, avoid using such functions in queries that involve GROUP BY and ORDER BY.

Do not specify **CARDINALITY** for external scalar functions.

NO COLLID or COLLID *collection-id*
Identifies the package collection that is to be used when the function is executed. This is the package collection into which the DBRM that is associated with the function is bound.

NO COLLID
Specifies the package collection for the function is the same as the package collection of the program that invokes the function. If a trigger invokes the function, the collection of the trigger package is used. If the invoking program does not use a package, DB2 resolves the package by using the CURRENT PACKAGE PATH special register, the CURRENT PACKAGESET

special register, or the PKLIST bind option (in this order). For details about how DB2 uses these three items, see the information on package resolution in *DB2 Application Programming and SQL Guide*.

COLLID *collection-id*
Specifies the name of the package collection that is to be used when the function is executed.

WLM ENVIRONMENT
An SQL identifier that identifies the *name* of the WLM (workload manager) application environment in which the function is to run.

name
The WLM environment in which the function must run. If the user-defined function is nested and if the calling stored procedure or invoking user-defined function is not running in an address space associated with the specified WLM environment, DB2 routes the function request to a different address space.

(*name***,*)**
When an SQL application program calls the function, *name* specifies the WLM environment in which the function runs.

If another user-defined function or a stored procedure calls the function, the function runs in the same environment that the calling routine uses. In this case, authorization to run the function in the WLM environment is not checked because the authorization of the calling routine suffices.

The *name* of the WLM environment is an SQL identifier.

To change the environment in which the function is to run, you must have appropriate authority for the WLM environment. For an example of a RACF command that provides this authorization, see Running stored procedures.

ASUTIME
Specifies the total amount of processor time, in CPU service units, that a single invocation of the function can run. The value is unrelated to the ASUTIME column of the resource limit specification table.

When you are debugging a function, setting a limit can be helpful if the function gets caught in a loop. For information on service units, see *z/OS MVS Initialization and Tuning Guide*.

NO LIMIT
There is no limit on the service units.

LIMIT *integer*
The limit on the number of CPU service units is a positive *integer* in the range of 1 to 2 147 483 647. If the procedure uses more service units than the specified value, DB2 cancels the procedure. The CPU cycles that are consumed by parallel tasks in a procedure do not contribute towards the specified ASUTIME LIMIT.

STAY RESIDENT
Specifies whether the load module for the function is to remain resident in memory when the function ends.

NO The load module is deleted from memory after the function ends. Use **NO** for non-reentrant functions.

YES
The load module remains resident in memory after the function ends. Use **YES** for reentrant functions.

PROGRAM TYPE
Specifies whether the function program runs as a main routine or a subroutine.

SUB
The function runs as a subroutine.

MAIN
The function runs as a main routine.

Do not specify **PROGRAM TYPE MAIN** when **LANGUAGE JAVA** is in effect.

SECURITY
Specifies how the function interacts with an external security product, such as RACF, to control access to non-SQL resources.

DB2
The function does not require an external security environment. If the function accesses resources that an external security product protects, the access is performed using the authorization ID associated with the WLM-established stored procedure address space.

USER
An external security environment should be used with the function. If the function accesses resources that the external security product protects, the access is performed using the primary authorization ID of the process that invoked the function.

DEFINER
An external security environment should be used with the function. If the function accesses resources that the external security product protects, the access is performed using the authorization ID of the owner of the function.

STOP AFTER SYSTEM DEFAULT FAILURES, STOP AFTER *nn* **FAILURES, or CONTINUE AFTER FAILURE**
Specifies whether the routine is to be put in a stopped state after some number of failures. The following options must not be specified for SQL functions or sourced functions.

STOP AFTER SYSTEM DEFAULT FAILURES
Specifies that this routine should be placed in a stopped state after the number of failures indicated by the value of field MAX ABEND COUNT on installation panel DSNTIPX.

STOP AFTER *nn* **FAILURES**
Specifies that this routine should be placed in a stopped state after *nn* failures. The value *nn* can be an integer from 1 to 32767.

CONTINUE AFTER FAILURE
Specifies that this routine should not be placed in a stopped state after any failure.

RUN OPTIONS *run-time-options*
Specifies the Language Environment run time options to be used for the function. You must specify *run-time-options* as a character string that is no longer than 254 bytes. To replace any existing run time options with no options, specify an empty string with **RUN OPTIONS**. When you specify an

empty string, DB2 does not pass any run time options to Language Environment, and Language Environment uses its installation defaults.

For a description of the Language Environment run time options, see *z/OS Language Environment Programming Reference*.

Do not specify **RUN OPTIONS** when **LANGUAGE JAVA** is specified.

INHERIT SPECIAL REGISTERS or DEFAULT SPECIAL REGISTERS
Specifies how special registers are set on entry to the routine.

INHERIT SPECIAL REGISTERS
Specifies that special registers should be inherited according to the rules listed in the table for characteristics of special registers in a user-defined function in "Special registers in a user-defined function or a stored procedure" on page 205.

DEFAULT SPECIAL REGISTERS
Specifies that special registers should be initialized to the default values, as indicated by the rules in the table for characteristics of special registers in a user-defined function in "Special registers in a user-defined function or a stored procedure" on page 205.

STATIC DISPATCH
At function resolution time, DB2 chooses a function based on the static (or declared) types of the function parameters.

SECURED or NOT SECURED
Specifies whether the function is considered secure.

SECURED
Specifies that the function is considered secure.

NOT SECURED
Specifies that the function is considered not secure. NOT SECURED must not be specified when a row permission or a column mask depends on the function.

When the function is invoked, the arguments of the function must not reference a column for which a column mask is enabled when the table is using active column access control.

Notes

Invalidation of packages:
When an external function is altered, all the packages that refer to that function are marked invalid.

LANGUAGE C and the PARAMETER VARCHAR clause:
The ALTER statement does not allow you to alter the value of the **PARAMETER VARCHAR** or **PARAMETER CCSID** clauses that are associated with the function definition. However, you can alter the **LANGUAGE** clause for the function. If the **PARAMETER VARCHAR** clause is specified for the creation of a LANGUAGE C function, the catalog information for that option is not affected by a subsequent ALTER function statement. The function might be changed to a language other than C, in which case the **PARAMETER VARCHAR** setting is ignored. If the function is later changed back to LANGUAGE C, the setting of the **PARAMETER VARCHAR** option that was specified during the CREATE FUNCTION statement will be used.

Altering a function from NOT SECURED to SECURED:
Typically, the security administrator will examine the data that is accessed by a function, ensure that it is secure, and grant the CREATE_SECURE_OBJECT privilege to the user that requires privileges to change the user-defined function to be secured. After the function is changed to SECURED, the security administrator will revoke the CREATE_SECURE_OBJECT privilege from the owner of the function.

The function is considered secure after the ALTER FUNCTION statement is executed. DB2 treats the SECURED attribute as an assertion that declares that the security administrator has established an audit procedure for all changes to the user-defined function. DB2 assumes that such a control audit procedure is in place for all subsequent ALTER FUNCTION statements or changes to external packages.

Packages and statements in the dynamic statement cache that reference the function are invalidated.

Altering a function from SECURED to NOT SECURED:
Packages and statements in the dynamic statement cache that reference the function are invalidated when the function is changed from SECURED to NOT SECURED. An function that is not secured might negatively impact performance if that function accesses data in a table that is using row access control or column access control. To minimize the performance impact, either change the function to use the SECURED option or deactivate row access control or column access control for the table that the function is accessing.

Invoking other user-defined functions in a secure function:
When a secure user-defined function is referenced in an SQL data change statement that references a table that is using row access control or column access control, and if the secure user-defined function invokes other user-defined functions, the nested user-defined functions are not validated as secure. If those nested functions can access sensitive data, the security administrator needs to ensure that those functions are allowed to access sensitive data and should ensure that a change control audit procedure has been established for all changes to those functions.

The SECURE column in the DSN_FUNCTION_TABLE EXPLAIN table:
The SECURE column in the DSN_FUNCTION_TABLE EXPLAIN table indicates if a user-defined function is considered secure.

Alternative syntax and synonyms:
To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords:
- VARIANT as a synonym for **NOT DETERMINISTIC**
- NOT VARIANT as a synonym for **DETERMINISTIC**
- NOT NULL CALL as a synonym for **RETURNS NULL ON NULL INPUT**
- NULL CALL as a synonym for **CALLED ON NULL INPUT**
- PARAMETER STYLE DB2SQL as a synonym for **PARAMETER STYLE SQL**
- TIMEZONE can be specified as an alternative to TIME ZONE.

Examples

Example 1: Assume that two functions CENTER are in the PELLOW schema. The first function has two input parameters with INTEGER and FLOAT data types,

respectively. The specific name for the first function is FOCUS1. The second function has three parameters with CHAR(25), DEC(5,2), and INTEGER data types.

Using the specific name to identify the function, change the WLM environment in which the first function runs from WLMENVNAME1 to WLMENVNAME2:

```
ALTER SPECIFIC FUNCTION ENGLES.FOCUS1 WLM ENVIRONMENT WLMENVNAME2;
```

Example 2: Change the second function that is described in *Example 1* so that it is not invoked when any of the arguments are null. Use the function signature to identify the function:

```
ALTER FUNCTION ENGLES.CENTER (CHAR(25), DEC(5,2), INTEGER)
   RETURNS NULL ON NULL INPUT;
```

You can also code the ALTER FUNCTION statement without the exact values for the CHAR and DEC data types:

```
ALTER FUNCTION ENGLES.CENTER (CHAR(), DEC(), INTEGER)
   RETURNS NULL ON NULL INPUT;
```

If you use empty parentheses, DB2 is to ignore the length, precision, and scale attributes when looking for matching data types to find the function.

ALTER FUNCTION (SQL scalar)

The ALTER FUNCTION (SQL scalar) statement changes the description of a user-defined SQL scalar function at the current server.

Invocation

For an inline SQL function, this statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

For a non-inline SQL function, this statement can only be dynamically prepared and the DYNAMICRULES run behavior must be specified implicitly or explicitly.

Authorization

The privilege set defined below must include at least one of the following:

- Ownership of the function
- The ALTERIN privilege on the schema
- SYSADM authority
- SYSCTRL authority
- System DBADM

The authorization ID that matches the schema name implicitly has the ALTERIN privilege on the schema.

If the authorization ID that is used to alter the function has installation SYSADM authority, the function is identified as system-defined function when the function definition is reevaluated.

Additional privileges might be required in the following situations:

- If *SQL-routine-body* is specified, the privilege set must include the privileges that are required to execute the statements in *SQL-routine-body*.
- If a distinct type is referenced (i.e. as the data type of an SQL variable in the body of the function), the privilege set must include at least one of the following:
  - Ownership of the distinct type
  - The USAGE privilege on the distinct type
  - SYSADM authority
- If the function uses a table as a parameter, the privilege set must also include at least one of the following:
  - Ownership of the table
  - The SELECT privilege on the table
  - SYSADM authority
- If the WLM ENVIRONMENT FOR DEBUG MODE clause is specified, the privilege set must include the authority to define programs that run in the specified WLM environment. This authorization is obtained from an external security product, such as RACF.
- When replacing an SQL scalar function, the privilege set must include the required authorization to add a new package or a new version of an existing

package depending on the value of the BIND NEW PACKAGE field on installation panel DSNTIPP, or the privilege set must include SYSADM or SYSCTRL authority.

At least one of the following privileges is required if the SECURED option is specified or if the function is currently secured and the NOT SECURED option is specified:

- SECADM authority
- CREATE_SECURE_OBJECT privilege

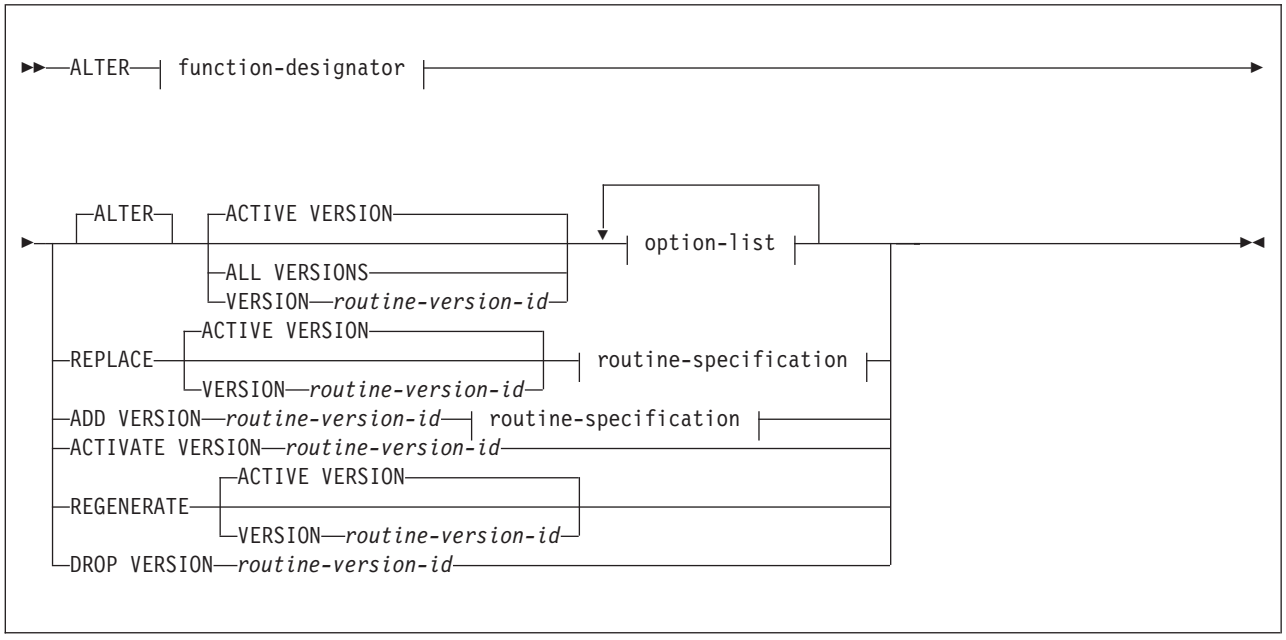
At least one of those privileges is also required if the function is currently secure and the ALTER ACTIVE VERSION, ALTER VERSION *routine-version-id*, ADD VERSION, or REPLACE clause is specified.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the package.

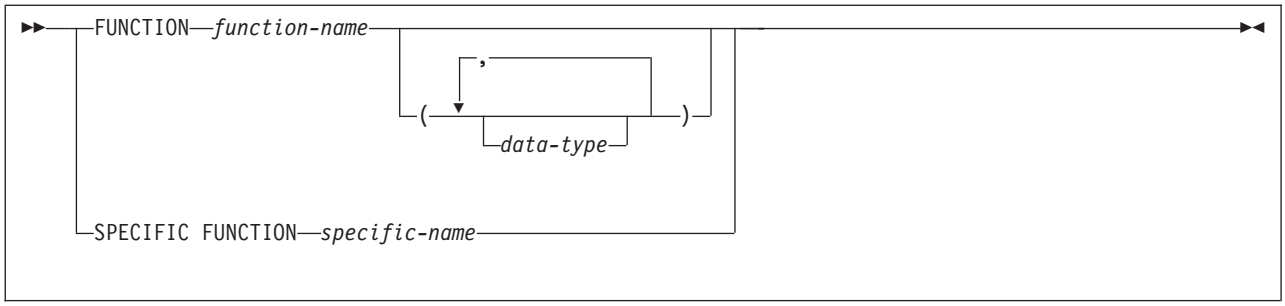
If the statement is dynamically prepared, the privilege set is the set of privileges that are held by the SQL authorization IDs of the process. The specified routine name can include a schema name (a qualifier). However, if the schema name is not the same as one of these SQL authorization IDs, one of the following conditions must be met:

- The privilege set includes SYSADM authority
- The privilege set includes SYSCTRL authority
- The SQL authorization ID of the process has the ALTERIN privilege on the schema

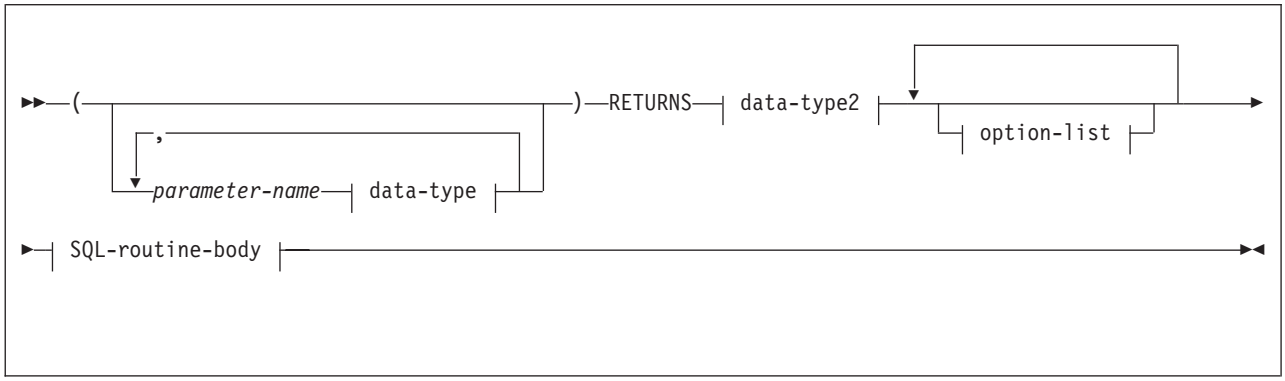
Syntax



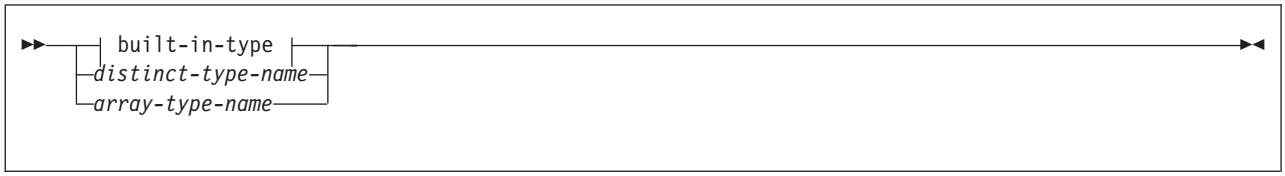
function-designator:



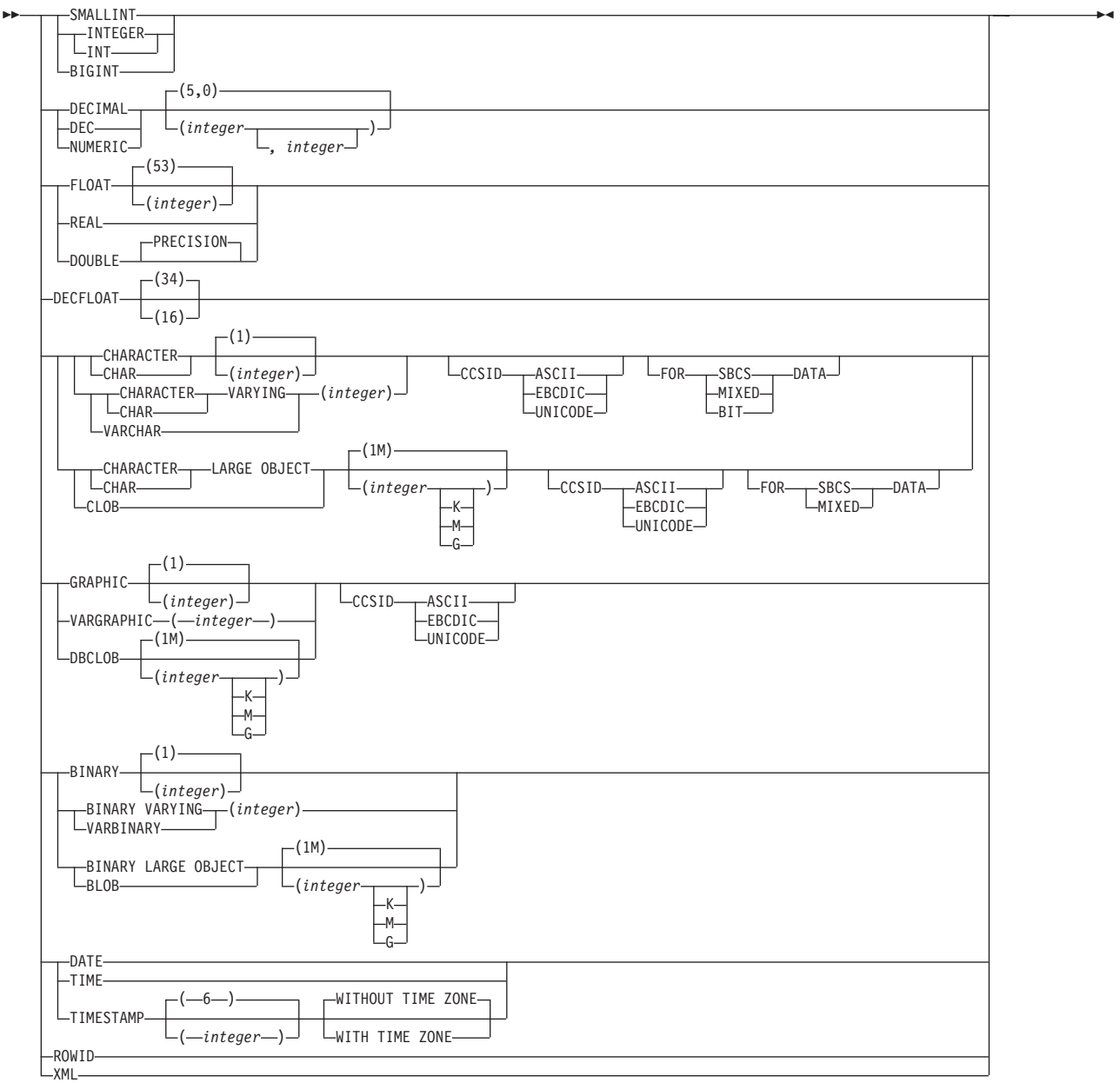
routine-specification:



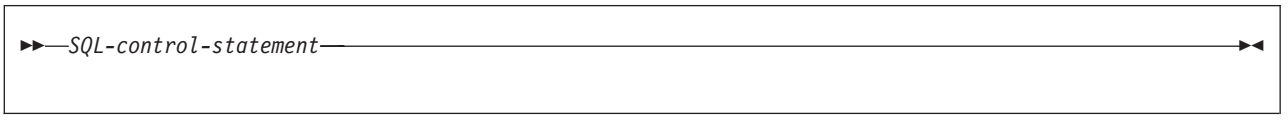
data-type, data-type2:



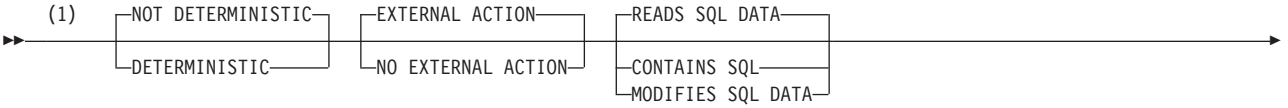
built-in-type:

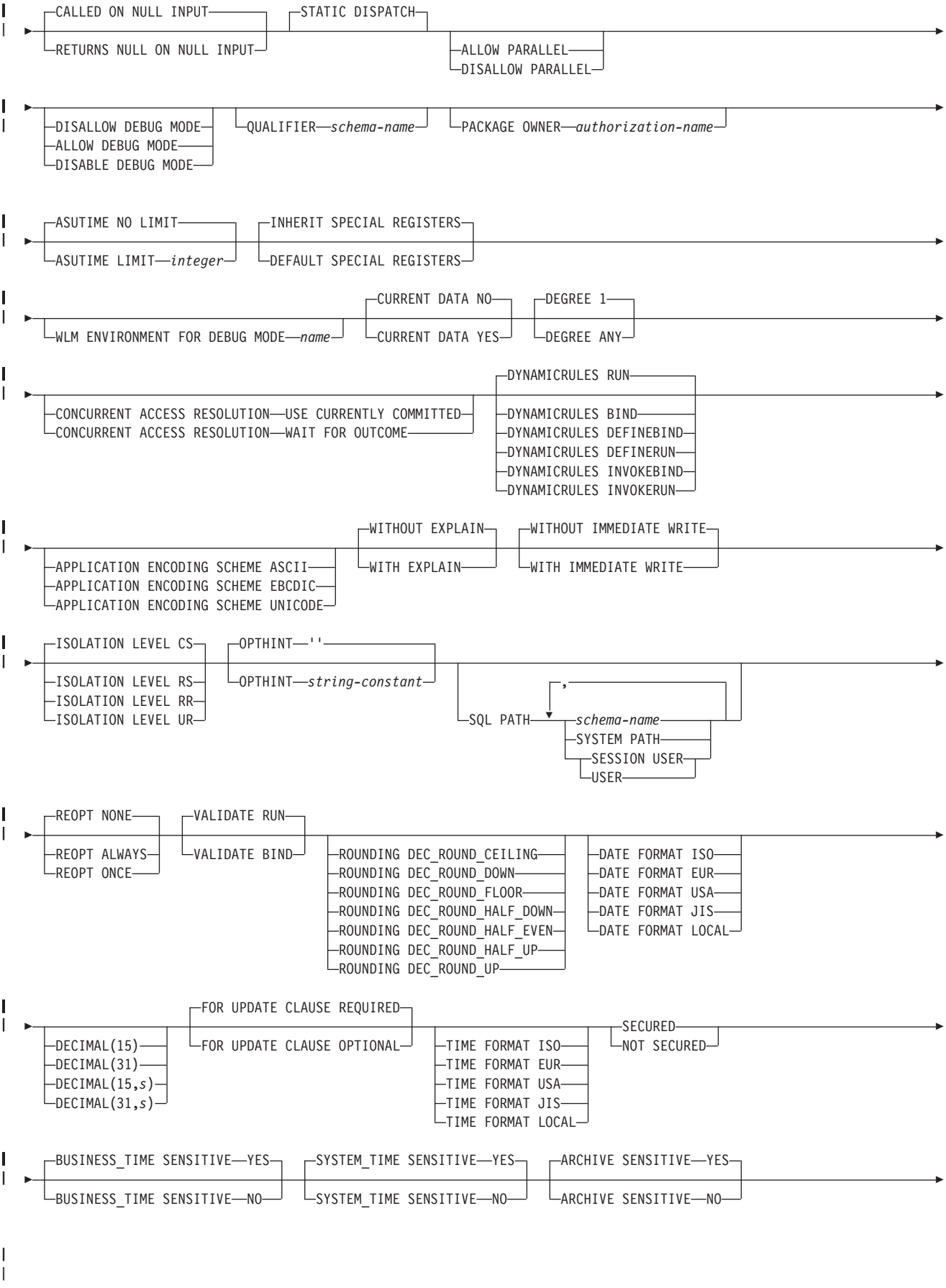


SQL-routine-body:



option-list: (Specify options in any order. Specify at least one option. Do not specify the same option more than one time.)




CALLED ON NULL INPUT
RETURNS NULL ON NULL INPUT
STATIC DISPATCH
ALLOW PARALLEL
DISALLOW PARALLEL
DISALLOW DEBUG MODE
ALLOW DEBUG MODE
DISABLE DEBUG MODE
QUALIFIER schema-name
PACKAGE OWNER authorization-name
ASUTIME NO LIMIT
ASUTIME LIMIT integer
INHERIT SPECIAL REGISTERS
DEFAULT SPECIAL REGISTERS
WLM ENVIRONMENT FOR DEBUG MODE name
CURRENT DATA NO
CURRENT DATA YES
DEGREE 1
DEGREE ANY
CONCURRENT ACCESS RESOLUTION USE CURRENTLY COMMITTED
CONCURRENT ACCESS RESOLUTION WAIT FOR OUTCOME
DYNAMICRULES RUN
DYNAMICRULES BIND
DYNAMICRULES DEFINEBIND
DYNAMICRULES DEFINERUN
DYNAMICRULES INVOKEBIND
DYNAMICRULES INVOKERUN
APPLICATION ENCODING SCHEME ASCII
APPLICATION ENCODING SCHEME EBCDIC
APPLICATION ENCODING SCHEME UNICODE
WITHOUT EXPLAIN
WITH EXPLAIN
WITHOUT IMMEDIATE WRITE
WITH IMMEDIATE WRITE
ISOLATION LEVEL CS
ISOLATION LEVEL RS
ISOLATION LEVEL RR
ISOLATION LEVEL UR
OPTHINT ''
OPTHINT string-constant
SQL PATH
,
schema-name
SYSTEM PATH
SESSION USER
USER
REOPT NONE
REOPT ALWAYS
REOPT ONCE
VALIDATE RUN
VALIDATE BIND
ROUNDING DEC_ROUND_CEILING
ROUNDING DEC_ROUND_DOWN
ROUNDING DEC_ROUND_FLOOR
ROUNDING DEC_ROUND_HALF_DOWN
ROUNDING DEC_ROUND_HALF_EVEN
ROUNDING DEC_ROUND_HALF_UP
ROUNDING DEC_ROUND_UP
DATE FORMAT ISO
DATE FORMAT EUR
DATE FORMAT USA
DATE FORMAT JIS
DATE FORMAT LOCAL
FOR UPDATE CLAUSE REQUIRED
FOR UPDATE CLAUSE OPTIONAL
DECIMAL(15)
DECIMAL(31)
DECIMAL(15,s)
DECIMAL(31,s)
TIME FORMAT ISO
TIME FORMAT EUR
TIME FORMAT USA
TIME FORMAT JIS
TIME FORMAT LOCAL
SECURED
NOT SECURED
BUSINESS_TIME SENSITIVE YES
BUSINESS_TIME SENSITIVE NO
SYSTEM_TIME SENSITIVE YES
SYSTEM_TIME SENSITIVE NO
ARCHIVE SENSITIVE YES
ARCHIVE SENSITIVE NO

Notes:

1 Only LANGUAGE SQL, NOT DETERMINISTIC, DETERMINISTIC, EXTERNAL ACTION, NO EXTERNAL ACTION, CONTAINS SQL, READS SQL DATA, STATIC DISPATCH, and CALLED ON NULL INPUT can be specified for an inline SQL scalar function.

Description

One of the following three clauses identifies the function to be changed.

FUNCTION *function-name*
Identifies the SQL function by its function name.

The identified function must be an SQL scalar function. There must be exactly one function with *function-name* in the schema. The function can have any number of input parameters.[24] If the schema does not contain a function with *function-name* or contains more than one function with this name, an error occurs.

FUNCTION *function-name (parameter-type,...)*
Identifies the SQL function by its function signature, which uniquely identifies the function.

function-name
Gives the function name of the SQL function.

If the function is an inline function, the only options that can be specified are: LANGUAGE SQL, NOT DETERMINISTIC, DETERMINISTIC, EXTERNAL ACTION, NO EXTERNAL ACTION, CONTAINS SQL, READS SQL DATA, STATIC DISPATCH, and CALLED ON NULL INPUT.

If *function-name*() is specified, the function that is identified must have zero parameters.

(*parameter-type***,...)**
Specifies the number of input parameters of the function and the name and data type of each parameter.

If the function was defined with a table parameter (the **LIKE TABLE** *name* **AS LOCATOR** clause was specified in the CREATE FUNCTION statement to indicate that one of the input parameters is a transition table), the function signature cannot be used to uniquely identify the function. Instead, use one of the other syntax variations to identify the function with its function name, if unique, or its specific name.

(*data-type***,...)**
Identifies the number of input parameters of the function and the data type of each parameter. The data type of each parameter must match the data type that was specified in the CREATE FUNCTION statement for the parameter in the corresponding position. The number of data types and the logical concatenation of the data types are used to uniquely identify the function. Therefore, you cannot change the number of parameters or the data types of the parameters.

24. If the function has more than 30 parameters, only the first 30 parameters are used to determine whether the function is unique.

For data types that have a length, precision, or scale attribute, you can use a set of empty parentheses, specify a value, or accept the default values:

- Empty parentheses indicate that DB2 is to ignore the attribute when determining whether the data types match.

  For example, DEC() will be considered a match for a parameter of a function defined with a data type of DEC(7,2). Similarly DECFLOAT() will be considered a match for DECFLOAT(16) or DECFLOAT(34).

  FLOAT cannot be specified with empty parentheses because its parameter value indicates different data types (REAL or DOUBLE).
- If you use a specific value for a length, precision, or scale attribute, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

  The specific value for FLOAT(*n*) does not have to exactly match the defined value of the source function because 1<=*n*<= 21 indicates REAL and 22<=*n*<=53 indicates DOUBLE. Matching is based on whether the data type is REAL or DOUBLE.
- If length, precision, or scale is not explicitly specified and empty parentheses are not specified, the default length of the data type is implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

For data types with a subtype or encoding scheme attribute, specifying the **FOR** *subtype* **DATA** clause or the **CCSID** clause is optional. Omission of either clause indicates that DB2 is to ignore the attribute when determining whether the data types match. If you specify either clause, it must match the value that was implicitly or explicitly specified in the CREATE FUNCTION statement.

See "CREATE FUNCTION" on page 1165 for more information on the specification of the parameter list.

A function with the function signature must exist in the explicitly or implicitly specified schema.

SPECIFIC FUNCTION *specific-name*
Identifies a particular user-defined function by its specific name. The name is implicitly or explicitly qualified with a schema name. A function with the specific name must exist in the schema. If the specific name is not qualified, it is implicitly qualified with a schema name as described in the description for **FUNCTION** *function-name*

ALTER ACTIVE VERSION, ALL VERSIONS, or VERSION *routine-version-id*
Specifies that a version of the function is to be changed. When you change a function using **ALTER** option-list, any option that is not explicitly specified will use the existing value from the version of the function that is being changed.

ACTIVE VERSION, ALL VERSION or, VERSION *routine-version-id*
Identifies the version of the function that is to be changed.

ACTIVE VERSION
Specifies that the currently active version of the function is to be changed, replaced, or regenerated. If the function is secure, the changed, replaced, or regenerated version remains secure.

ACTIVE VERSION is the default.

ALL VERSIONS
Specifies that all of the versions of the function are to be changed. SECURED and NOT SECURED are the only options that can be changed when ALL VERSIONS is specified.

VERSION *routine-version-id*
Identifies the version of the function that is to be changed, replaced, or regenerated. *routine-version-id* is the version identifier that is assigned when the version of the function is defined. *routine-version-id* must identify a version of the specified function that exists at the current server. If the function is secure, the changed, replaced, or regenerated version remains secure.

REPLACE ACTIVE VERSION or VERSION *routine-version-id*
Specifies that a version of the function is to be replaced.

Binding the replaced version of the function might result in a new access path even if the routine body is not being changed.

When you replace a function, the data types, CCSID specifications, and character data attributes (FOR BIT/SBCS/MIXED DATA) of the parameters must be the same as the attributes of the corresponding parameters for the currently active version of the function. For options that are not explicitly specified, the system default values for those options are used, even if those options were explicitly specified for the version of the function that is being replaced. This is not the case for versions of the function that specified DISABLE DEBUG MODE. If DISABLE DEBUG MODE is specified for a version of a function, it cannot be changed by using the REPLACE clause. When a function definition is replaced, any existing comments in the catalog for that definition of the function are removed.

ACTIVE VERSION or VERSION *routine-version-id*
Identifies the version of the function that is to be replaced.

ACTIVE VERSION
Specifies that the currently active version of the function is to be changed, replaced, or regenerated. If the function is secure, the changed, replaced, or regenerated version remains secure.

ACTIVE VERSION is the default.

VERSION *routine-version-id*
Identifies the version of the function that is to be changed, replaced, or regenerated. *routine-version-id* is the version identifier that is assigned when the version of the function is defined. *routine-version-id* must identify a version of the specified function that exists at the current server. If the function is secure, the changed, replaced, or regenerated version remains secure.

ADD VERSION *routine-version-id*
Specifies that a new version of the function is to be created. *routine-version-id* is the version identifier for the new version of the function. *routine-version-id* must not identify a version of the specified function that already exists at the current server.

When a new version of a function is created, the comment that is recorded in the catalog for the new version will be the same as the comment that is in the catalog for the currently active version.

When you add a new version of a function, the data types, CCSID specifications, and character data attributes (FOR BIT/SBCS/MIXED DATA) of

the parameters must be the same as the attributes of the corresponding parameters for the currently active version of the function. The parameter names can differ from the other versions of the function. For options that are not explicitly specified, the system default values will be used.

If the function is secure, the new version is considered secure.

ACTIVATE VERSION *routine-version-id*
Specifies the version of the function that is to be the currently active version. *routine-version-id* is the version identifier that is assigned when the version of the function is defined. The version that is specified with *routine-version-id* is the version that will be invoked by a function invocation. *routine-version-id* must identify a version of the function that exists at the current server.

REGENERATE ACTIVE VERSION or VERSION *routine-version-id*
Specifies that a version of the function is to be regenerated. When DB2 maintenance is applied that changes how an SQL function is generated, the function might need to be regenerated to process the changes from applying the maintenance.

REGENERATE automatically rebinds, at the current server, the package for the SQL control statements for the function and rebinds the package for the SQL statements that are included in the body of the function.

REGENERATE is different than the REBIND PACKAGE command. REBIND PACKAGE rebinds the SQL statements (usually to generate better access paths for those statement) but the SQL control statements in the function definition are not rebound.

When a function definition is regenerated, any existing comments in the catalog for that definition of the function are not removed.

ACTIVE VERSION or VERSION *routine-version-id*
Identifies the version of the function that is to be regenerated.

ACTIVE VERSION
Specifies that the currently active version of the function is to be changed, replaced, or regenerated. If the function is secure, the changed, replaced, or regenerated version remains secure.

ACTIVE VERSION is the default.

VERSION *routine-version-id*
Identifies the version of the function that is to be changed, replaced, or regenerated. *routine-version-id* is the version identifier that is assigned when the version of the function is defined. *routine-version-id* must identify a version of the specified function that exists at the current server. If the function is secure, the changed, replaced, or regenerated version remains secure.

DROP VERSION *routine-version-id*
Drops the version of the function that is identified with *routine-version-id*. *routine-version-id* is the version identifier that is assigned when the version is defined. *routine-version-id* must identify a version of the function that exists at the current server and must not identify the currently active version of the function. Only the identified version of the function is dropped.

When only a single version of the function exists at the current server, use the DROP FUNCTION statement to drop the function.

RETURNS
Identifies the output of the function.

data-type2
Specifies the data type of the output. The data type must match the data type that was specified in the RETURNS clause of the CREATE FUNCTION statement.

NOT DETERMINISTIC or DETERMINISTIC
Specifies whether the function returns the same results each time that the function is invoked with the same input arguments.

NOT DETERMINISTIC
The function might not return the same result each time that the function is invoked with the same input arguments. The function depends on some state values that affect the results. DB2 uses this information to disable the merging of views and table expressions when processing SELECT or SQL data change statements that refer to this function. An example of a function that is not deterministic is one that generates random numbers.

NOT DETERMINISTIC must be specified explicitly or implicitly if the function program accesses a special register or invokes another function that is not deterministic.

DETERMINISTIC
The function always returns the same result each time that the function is invoked with the same input arguments. An example of a deterministic function is a function that calculates the square root of the input. DB2 uses this information to enable the merging of views and table expressions for SELECT or SQL data change statements that refer to this function. If applicable, specify **DETERMINISTIC** to prevent non-optimal access paths from being chosen for SQL statements that refer to this function.

DB2 does not verify that the function program is consistent with the specification of **DETERMINISTIC** or **NOT DETERMINISTIC**.

EXTERNAL ACTION or NO EXTERNAL ACTION
Specifies whether the function takes an action that changes the state of an object that DB2 does not manage. An example of an external action is sending a message or writing a record to a file.

EXTERNAL ACTION
The function can take an action that changes the state of an object that DB2 does not manage.

Some SQL statements that invoke functions with external actions can result in incorrect results if parallel tasks execute the function. For example, if the function sends a note for each initial call to it, one note is sent for each parallel task instead of once for the function.

If you specify **EXTERNAL ACTION**, DB2:

- Materializes the views and table expressions in SELECT or SQL data change statements that refer to the function. This materialization can adversely affect the access paths that are chosen for the SQL statements that refer to this function. Do not specify **EXTERNAL ACTION** if the function does not have an external action.
- Does not move the function from one task control block (TCB) to another between FETCH operations.
- Does not allow another function or stored procedure to use the TCB until the cursor is closed. This is also applicable for cursors declared WITH HOLD.

The only changes to resources made outside of DB2 that are under the control of commit and rollback operations are those changes made under RRS control.

EXTERNAL ACTION must be specified implicitly or explicitly specified if the SQL routine body invokes a function that is defined with **EXTERNAL ACTION**.

NO EXTERNAL ACTION
The function does not take any action that changes the state of an object that DB2 does not manage. DB2 uses this information to enable the merging of views and table expressions for SELECT or SQL data change statements that refer to this function. If applicable, specify **NO EXTERNAL ACTION** to prevent non-optimal access paths from being chosen for SQL statements that refer to this function.

DB2 does not verify that the function program is consistent with the specification of **EXTERNAL ACTION** or **NO EXTERNAL ACTION**.

MODIFIES SQL DATA, READS SQL DATA, or CONTAINS SQL
Specifies which SQL statements, if any, can be executed in the function or any routine that is called from this function.

MODIFIES SQL DATA
Specifies that the function can execute any SQL statement except the statements that are not supported in functions. Do not specify MODIFIES SQL DATA when ALLOW PARALLEL is in effect.

READS SQL DATA
Specifies that the function can execute statements with a data access classification of READS SQL DATA, CONTAINS SQL, or NO SQL. The function cannot execute SQL statements that modify data.

READS SQL DATA is the default.

CONTAINS SQL
Specifies that the function can execute only SQL statements with a data access classification of CONTAINS SQL or NO SQL. The function cannot execute SQL statements the read or modify data.

CALLED ON NULL INPUT or RETURNS NULL ON NULL INPUT
Specifies whether the function is invoked if any of the input arguments is null at execution time.

CALLED ON NULL INPUT
Specifies that the function is to be invoked, if any, or all, argument values are null. This specification means that the body of the function must be coded to test for null argument values.

CALLED ON NULL INPUT is the default.

RETURNS NULL ON NULL INPUT
Specifies that the function is not invoked and returns the null value if any of the input arguments is null.

STATIC DISPATCH
At function resolution time, DB2 chooses a function based on the static (or declared) types of the function parameters.

STATIC DISPATCH is the default.

ALLOW PARALLEL or DISALLOW PARALLEL
Specifies if the function can be run in parallel. The default is DISALLOW PARALLEL, if you specify one or more of the following clauses:

- NOT DETERMINISTIC
- EXTERNAL ACTION
- MODIFIES SQL DATA

Otherwise, ALLOW PARALLEL is the default.

ALLOW PARALLEL
Specifies that the function can be run in parallel.

DISALLOW PARALLEL
Specifies that the function cannot be run in parallel.

ALLOW DEBUG MODE, DISALLOW DEBUG MODE, or DISABLE DEBUG MODE
Specifies whether this version of the routine can be run in debugging mode. The default is determined using the value of the CURRENT DEBUG MODE special register.

ALLOW DEBUG MODE
Specifies that this version of the routine can be run in debugging mode. When this version of the routine is invoked and debugging is attempted, a WLM environment must be available.

DISALLOW DEBUG MODE
Specifies that this version of the routine cannot be run in debugging mode.

You can use an ALTER statement to change this option to ALLOW DEBUG MODE for this initial version of the routine.

DISABLE DEBUG MODE
Specifies that this version of the routine can never be run in debugging mode.

This version of the routine cannot be changed to specify ALLOW DEBUG MODE or DISALLOW DEBUG MODE after this version of the routine has been created or altered to use DISABLE DEBUG MODE. To change this option, drop the routine and create it again using the option that you want. An alternative to dropping and recreating the routine is to create a version of the routine that uses the option that you want and making that version the active version.

When DISABLE DEBUG MODE is in effect, the WLM ENVIRONMENT FOR DEBUG MODE is ignored.

QUALIFIER *schema-name*
Specifies the implicit qualifier that is used for unqualified names of tables, views, indexes, and aliases that are referenced in the routine body. The default value is the same as the default schema.

PACKAGE OWNER *authorization-name*
Specifies the owner of the package that is associated with the first version of the routine. The SQL authorization ID of the process is the default value.

The authorization ID must have the privileges that are required to execute the SQL statements that are contained in the routine body and must contain the necessary bind privileges. The value of PACKAGE OWNER is subject to translation when it is sent to a remote system.

If the privilege set lacks SYSADM or SYSCTRL authority, *authorization-name* must be the same as one of the authorization IDs of the process or the

authorization ID of the process. If the privilege set includes SYSADM or SYSCTRL authority, *authorization-name* can be any authorization ID that contains the necessary bind privileges.

ASUTIME
Specifies the total amount of processor time, in CPU service units, that a single invocation of a routine can run. The value is unrelated to the ASUTIME column of the resource limit specification table.

When you are debugging a routine, setting a limit can be helpful in case the routine gets caught in a loop. For information on service units, see *z/OS MVS Initialization and Tuning Guide*.

NO LIMIT
Specifies that there is no limit on the service units.

NO LIMIT is the default.

LIMIT *integer*
The limit on the number of CPU service units is a positive *integer* in the range of 1 to 2 147 483 647. If the procedure uses more service units than the specified value, DB2 cancels the procedure. The CPU cycles that are consumed by parallel tasks in a procedure do not contribute towards the specified ASUTIME LIMIT.

INHERIT SPECIAL REGISTERS or DEFAULT SPECIAL REGISTERS
Specifies how special registers are set on entry to the routine.

INHERIT SPECIAL REGISTERS
Specifies that the values of special registers are inherited, according to the rules that are listed in the table for characteristics of special registers in a routine in Table 40 on page 205.

INHERIT SPECIAL REGISTERS is the default.

DEFAULT SPECIAL REGISTERS
Specifies that special registers are initialized to the default values, as indicated by the rules in the table for characteristics of special registers in a routine in Table 40 on page 205.

WLM ENVIRONMENT FOR DEBUG MODE *name*
Specifies the WLM (workload manager) application environment that is used by DB2 when debugging the routine. The *name* of the WLM environment is an SQL identifier.

If you do not specify WLM ENVIRONMENT FOR DEBUG MODE, DB2 uses the default WLM-established stored procedure address space specified at installation time.

To define a routine that is to run in a specified WLM application environment, you must have the appropriate authority for the WLM application environment. For an example of a RACF command that provides this authorization, see Running stored procedures.

The WLM ENVIRONMENT FOR DEBUG MODE value is ignored when DISABLE DEBUG MODE is in effect.

CURRENT DATA(YES) or CURRENT DATA(NO)
Specifies whether to require data currency for read-only and ambiguous cursors when the isolation level of cursor stability is in effect. CURRENT DATA also determines whether block fetch can be used for distributed, ambiguous cursors.

YES
Specifies that data currency is required for read-only and ambiguous cursors. DB2 acquired page or row locks to ensure data currency. Block fetch is ignored for distributed, ambiguous cursors.

NO Specifies that data currency is not required for read-only and ambiguous cursors. Block fetch is allowed for distributed, ambiguous cursors. Use of **CURRENT DATA**(**NO**) is not recommended if the routine attempts to dynamically prepare and execute a DELETE WHERE CURRENT OF statement against an ambiguous cursor after that cursor is opened. You receive a negative SQLCODE if your routine attempts to use a DELETE WHERE CURRENT OF statement for any of the following cursors:

- A cursor that is using block fetch
- A cursor that is using query parallelism
- A cursor that is positioned on a row that is modified by this or another application process

NO is the default.

DEGREE
Specifies whether to attempt to run a query using parallel processing to maximize performance.

1 Specifies that parallel processing should not be used.

1 is the default.

ANY
Specifies that parallel processing can be used.

CONCURRENT ACCESS RESOLUTION
Specifies the whether processing uses only committed data or whether it will wait for commit or rollback of data that is in the process of being updated.

WAIT FOR OUTCOME
Specifies that processing will wait for the commit or rollback of data that is in the process of being updated.

USE CURRENTLY COMMITTED
Specifies that processing use the currently committed version of the data when data that is in the process of being updated is encountered. **USE CURRENTLY COMMITTED** is applicable on scans that access tables that are defined in universal table spaces with row or page level lock size.

When there is lock contention between a read transaction and an insert transaction, **USE CURRENTLY COMMITTED** is applicable to scans with isolation level CS or RS. Applicable scans include intent read scans for read-only and ambiguous queries and for updatable cursors. **USE CURRENTLY COMMITTED** is also applicable to scans initiated from WHERE predicates of UPDATE or DELETE statements and the subselect of INSERT statements.

When there is lock contention is between a read transaction and a delete transaction, **USE CURRENTLY COMMITTED** is applicable to scans with isolation level CS and when CURRENTDATA(NO) is specified.

DYNAMICRULES
Specifies the values that apply, at run time, for the following dynamic SQL attributes:

- The authorization ID that is used to check authorization
- The qualifier that is used for unqualified objects

- The source for application programming options that DB2 uses to parse and semantically verify dynamic SQL statements

DYNAMICRULES also specifies whether dynamic SQL statements can include GRANT, REVOKE, ALTER, CREATE, DROP, and RENAME statements.

In addition to the value of the DYNAMICRULES clause, the run time environment of a routine controls how dynamic SQL statements behave at run time. The combination of the DYNAMICRULES value and the run time environment determines the value for the dynamic SQL attributes. That set of attribute values is called the dynamic SQL statement behavior. The following values can be specified:

RUN
Specifies that dynamic SQL statements are to be processed using run behavior.

RUN is the default.

BIND
Specifies that dynamic SQL statements are to be processed using bind behavior.

DEFINEBIND
Specifies that dynamic SQL statements are to be processed using either define behavior or bind behavior.

DEFINERUN
Specifies that dynamic SQL statements are to be processed using either define behavior or run behavior.

INVOKEBIND
Specifies that dynamic SQL statements are to be processed using either invoke behavior or bind behavior.

INVOKERUN
Specifies that dynamic SQL statements are to be processed using either invoke behavior or run behavior.

See "Authorization IDs and dynamic SQL" on page 75 for information on the effects of these options.

APPLICATION ENCODING SCHEME
Specifies the default encoding scheme for SQL variables in static SQL statements in the routine body. The value is used for defining an SQL variable in a compound statement if the CCSID clause is not specified as part of the data type, and the PARAMETER CCSID routine option is not specified.

ASCII
Specifies that the data is encoded using the ASCII CCSIDs of the server.

EBCDIC
Specifies that the data is encoded using the EBCDIC CCSIDs of the server.

UNICODE
Specifies that the data is encoded using the Unicode CCSIDs of the server.

See the ENCODING bind option in *DB2 Command Reference* for information about how the default for this option is determined.

WITH EXPLAIN or WITHOUT EXPLAIN
Specifies whether information will be provided about how SQL statements in the routine will execute.

WITHOUT EXPLAIN
Specifies that information will not be provided about how SQL statements in the routine will execute.

You can get EXPLAIN output for a statement that is embedded in a routine that is specified using WITHOUT EXPLAIN by embedding the SQL statement EXPLAIN in the routine body. Otherwise, the value of the EXPLAIN option applies to all explainable SQL statements in the routine body, and to the fullselect portion of any DECLARE CURSOR statements.

WITHOUT EXPLAIN is the default.

WITH EXPLAIN
Specifies that information will be provided about how SQL statements in the routine will execute. Information is inserted into the table *owner*.PLAN_TABLE. *owner* is the authorization ID of the owner of the routine. Alternatively, the authorization ID of the owner of the routine can have an alias as *owner*.PLAN_TABLE that points to the base table, PLAN_TABLE. *owner* must also have the appropriate SELECT and INSERT privileges on that table. WITH EXPLAIN does not obtain information for statements that access remote objects. PLAN_TABLE must have a base table and can have multiple aliases with the same table name, PLAN_TABLE, but have different schema qualifiers. It cannot be a view or a synonym and should exist before the version is added or replaced. In all inserts to *owner*.PLAN_TABLE, the value of QUERYNO is the statement number that is assigned by DB2.

The WITH EXPLAIN option also populates two optional tables if they exist: DSN_STATEMNT_TABLE and DSN_FUNCTION_TABLE. DSN_STATEMNT_TABLE contains an estimate of the processing cost for an SQL statement. See *DB2 Application Programming and SQL Guide* for more information. DSN_FUNCTION_TABLE contains information about function resolution. See *DB2 Application Programming and SQL Guide* for more information.

For a description of the tables that are populated by the WITH EXPLAIN option, see "EXPLAIN" on page 1642.

WITH IMMEDIATE WRITE or WITHOUT IMMEDIATE WRITE
Specifies whether immediate writes are to be done for updates that are made to group buffer pool dependent page sets or partitions. This option is only applicable for data sharing environments. The IMMEDWRITE subsystem parameter has no affect of this option. *DB2 Command Reference* shows the implied hierarchy of the IMMEDWRITE bind option (which is similar to this routine option) as it affects run time.

WITHOUT IMMEDIATE WRITE
Specifies that normal write activity is performed. Updated pages that are group buffer pool dependent are written at or before phase one of commit or at the end of abort for transactions that have been rolled back.

WITHOUT IMMEDIATE WRITE is the default.

WITH IMMEDIATE WRITE
Specifies that updated pages that are group buffer pool dependent are immediately written as soon as the buffer update completes. Updated pages are written immediately even if the buffer is updated during forward progress or during the rollback of a transaction. **WITH IMMEDIATE WRITE** might impact performance.

ISOLATION LEVEL RR, RS, CS, or UR
Specifies how far to isolate the routine from the effects of other running applications. For information about isolation levels, see *DB2 Performance Monitoring and Tuning Guide*.

RR Specifies repeatable read.

RS Specifies read stability.

CS Specifies cursor stability. **CS** is the default.

UR Specifies uncommitted read.

OPTHINT *string-constant*
Specifies whether query optimization hints are used for static SQL statements that are contained within the body of the routine.

string-constant is a character string of up to 128 bytes in length, which is used by the DB2 subsystem when searching the PLAN_TABLE for rows to use as input. The default value is an empty string, which indicates that the DB2 subsystem does not use optimization hints for static SQL statements.

Optimization hints are only used if optimization hints are enabled for your system. See *DB2 Installation Guide* for information about enabling optimization hints.

SQL PATH
Specifies the SQL path that the DB2 subsystem uses to resolve unqualified user-defined types, functions, and procedure names (in CALL statements) in the body of the routine. The maximum length of the SQL path is 2048 bytes. DB2 calculates the length by taking each *schema-name* that is specified and removing any trailing blanks from it, adding two delimiters around it, and adding one comma after each schema name except for the last name. The length of the resulting string cannot exceed 2048 bytes.

schema-name
Identifies a schema. DB2 does not verify that the schema exists when the ALTER statement is processed. The same schema name should not appear more than one time in the list of schema names.

SYSPUBLIC must not be specified for the SQL path.

SYSTEM PATH
Specifies the schema names "SYSIBM", "SYSFUN", "SYSPROC", "SYSIBMADM".

SESSION_USER or USER
Specifies the value of the SESSION_USER (or USER) special register. At the time the ALTER statement is processed, the actual length is included in the total length of the list of schema names that is specified for the SQL PATH option.

REOPT
Specifies if DB2 will determine the access path at run time by using the values of SQL variables or SQL parameters, parameter markers, and special registers.

NONE
Specifies that DB2 does not determine the access path at run time by using the values of SQL variables or SQL parameters, parameter markers, and special registers.

NONE is the default.

ALWAYS
Specifies that DB2 always determines the access path at run time each time an SQL statement is run. Do not specify REOPT ALWAYS with the WITH KEEP DYNAMIC or NODEFER PREPARE clauses.

ONCE
Specifies that DB2 determine the access path for any dynamic SQL statements only once, at the first time the statement is opened. This access path is used until the prepared statement is invalidated or removed from the dynamic statement cache and need to be prepared again.

VALIDATE RUN or VALIDATE BIND
Specifies whether to recheck, at run time, errors of the type "OBJECT NOT FOUND" and "NOT AUTHORIZED" that are found during bind or rebind. The option has no effect if all objects and needed privileges exist.

VALIDATE RUN
Specifies that if needed objects or privileges do not exist when the CREATE statement is processed, warning messages are returned, but the CREATE statement succeeds. The DB2 subsystem rechecks for the objects and privileges at run time for those SQL statements that failed the checks during processing of the CREATE statement. The authorization checks the use of the authorization ID of the owner of the routine.

VALIDATE RUN is the default.

VALIDATE BIND
Specifies that if needed objects or privileges do not exist at the time the CREATE statement is processed, an error is issued and the CREATE statement fails.

ROUNDING
Specifies the rounding mode for manipulation of DECFLOAT data. The default value is taken from the DEFAULT DECIMAL FLOATING POINT ROUNDING MODE in DECP.

DEC_ROUND_CEILING
Specifies numbers are rounded towards positive infinity.

DEC_ROUND_DOWN
Specifies numbers are rounded towards 0 (truncation).

DEC_ROUND_FLOOR
Specifies numbers are rounded towards negative infinity.

DEC_ROUND_HALF_DOWN
Specifies numbers are rounded to nearest; if equidistant, round down.

DEC_ROUND_HALF_EVEN
Specifies numbers are rounded to nearest; if equidistant, round so that the final digit is even.

DEC_ROUND_HALF_UP
Specifies numbers are rounded to nearest; if equidistant, round up.

DEC_ROUND_UP
Specifies numbers are rounded away from 0.

DATE FORMAT ISO, EUR, USA, JIS, or LOCAL
Specifies the date format for result values that are string representations of date or time values. See “String representations of datetime values” on page 101 for more information.

The default format is specified in the DATE FORMAT field of installation panel DSNTIP4 of the system where the routine is defined. You cannot use the LOCAL option unless you have a date exit routine.

DECIMAL(15), DECIMAL(31), DECIMAL(15,*s*), or DECIMAL(31,*s*)
Specifies the maximum precision that is to be used for decimal arithmetic operations. See "Arithmetic with two decimal operands" on page 244 for more information. The default format is specified in the DECIMAL ARITHMETIC field of installation panel DSNTIPF of the system where the routine is defined. If the form *pp.s* is specified, *s* must be a number between 1 and 9. *s* represents the minimum scale that is to be used for division.

FOR UPDATE CLAUSE OPTIONAL or FOR UPDATE CLAUSE REQUIRED
Specifies whether the FOR UPDATE clause is required for a DECLARE CURSOR statement if the cursor is to be used to perform positioned updates.

FOR UPDATE CLAUSE REQUIRED
Specifies that a FOR UPDATE clause must be specified as part of the cursor definition if the cursor will be used to make positioned updates.

FOR UPDATE CLAUSE REQUIRED is the default.

FOR UPDATE CLAUSE OPTIONAL
Specifies that the FOR UPDATE clause does not need to be specified in order for a cursor to be used for positioned updates. The routine body can include positioned UPDATE statements that update columns that the user is authorized to update.

The FOR UPDATE clause with no column list applies to static or dynamic SQL statements. Even if you do not use this clause, you can specify FOR UPDATE OF with a column list to restrict updates to only the columns that are named in the FOR UPDATE clause and to specify the acquisition of update locks.

TIME FORMAT ISO, EUR, USA, JIS, or LOCAL
Specifies the time format for result values that are string representations of date or time values. See "String representations of datetime values" on page 101 for more information.

The default format is specified in the TIME FORMAT field of installation panel DSNTIP4 of the system where the routine is defined. You cannot use the LOCAL option unless you have a date exit routine.

SECURED or NOT SECURED
Specifies if the function is considered secure. When the option is specified with the **ALL VERSIONS** clause, it applies to all existing versions and to any future versions of the function. When it is specified with other clauses such as **ADD VERSION**, or **REPLACE**, the value must be the same as the value that is in effect for the function that is being changed.

SECURED
Specifies that the function is considered secure.

NOT SECURED
Specifies that the function is considered not secure. NOT SECURED must not be specified when a row permission or a column mask depends on the function.

When the function is invoked, the arguments of the function must not reference a column for which a column mask is enabled when the table is using active column access control.

BUSINESS_TIME SENSITIVE
Determines whether references to application-period temporal tables in both static and dynamic SQL statements are affected by the value of the CURRENT TEMPORAL BUSINESS_TIME special register.

YES
References to application-period temporal tables are affected by the value of the CURRENT TEMPORAL BUSINESS_TIME special register. YES is the default value.

NO References to application-period temporal tables are not affected by the value of the CURRENT TEMPORAL BUSINESS_TIME special register.

Related information:

"CURRENT TEMPORAL BUSINESS_TIME" on page 194

SYSTEM_TIME SENSITIVE
Determines whether references to system-period temporal tables in both static and dynamic SQL statements are affected by the value of the CURRENT TEMPORAL SYSTEM_TIME special register.

YES
References to system-period temporal tables are affected by the value of the CURRENT TEMPORAL SYSTEM_TIME special register. YES is the default value.

NO References to system-period temporal tables are not affected by the value of the CURRENT TEMPORAL SYSTEM_TIME special register.

Related information:

"CURRENT TEMPORAL SYSTEM_TIME" on page 196

ARCHIVE SENSITIVE
Determines whether references to archive-enabled tables in SQL statements are affected by the value of the SYSIBMADM.GET_ARCHIVE global variable.

YES
References to archive-enabled tables are affected by the value of the SYSIBMADM.GET_ARCHIVE global variable. YES is the default value.

NO References to archive-enabled tables are not affected by the value of the SYSIBMADM.GET_ARCHIVE global variable.

Related information:

"References to built-in global variables" on page 223

APPLCOMPAT *compatibility-level*
Specifies the package compatibility level behavior for static SQL, If this option is not specified then the behavior is determined, in priority order, by the *compatibility-level* of the last BIND or REBIND of the package or the APPLCOMPAT system parameter. The following values of *compatibility-level* can be specified:

V10R1
The static SQL statements in the package have V10R1 compatibility behavior.

V11R1
The static SQL statements in the package have V11R1 compatibility behavior.

Related information:

APPL COMPAT LEVEL field (APPLCOMPAT subsystem parameter) (DB2 Installation and Migration)

SQL-routine-body

Specifies a single SQL statement, including a compound statement. See Chapter 6, "SQL control statements for SQL routines," on page 1963 for more information about defining SQL functions.

A call to a procedure that issues a COMMIT, ROLLBACK, CONNECT, RELEASE, or SET CONNECTION statement is not allowed in a function.

If the *SQL-routine-body* is a compound statement, it must contain at least one RETURN statement and a RETURN statement must be executed when the function is invoked.

An ALTER FUNCTION (SQL scalar) statement or an ALTER PROCEDURE (SQL native) statement with an ADD VERSION or REPLACE clause is not allowed in an *SQL-routine-body*.

Notes

ALTER FUNCTION for in use functions:

ALTER FUNCTION will be locked out from making changes if the function is in use. For example, if a query that is currently running is referencing an SQL scalar function named 'fn1' (*routine-version-id* is 'v1'), an ALTER FUNCTION fn1 ACTIVATE VERSION v2 statement will wait for the query that is currently running to complete before making 'v2' the active version for function 'fn1'. This wait for completion behavior happens even if the query invokes the function multiple times for processing multiple rows or if the query contains multiple references to the function that is being changed.

Considerations for changing a version of a function:

To change a version of a function, the environment settings that are in effect when the ALTER FUNCTION statement is issued must be the same as the environment settings that are in effect when the version of the function is first created using the CREATE FUNCTION or ALTER FUNCTION statement if one of the following options is specified:

- QUALIFIER
- PACKAGE OWNER
- WLM ENVIRONMENT FOR DEBUG MODE
- OPTHINT
- SQL PATH
- DECIMAL (if the value includes a comma)

Considerations for catalog comments for a routine definition:

When a function definition is replaced, any existing comment in the catalog for the definition is removed. However, when a function definition is regenerated, any existing comment in the catalog for the definition is retained.

Identifier resolution:

See Chapter 6, "SQL control statements for SQL routines," on page 1963 for information on how names are resolved to columns, SQL variables, or SQL parameters within an SQL routine.

If duplicate names are used for columns and SQL variables and parameters, qualify the duplicate names by using the table designator for columns, the routine name for parameters, and the label name for SQL variables.

Characteristics of the package that is generated for a function:

The package that is associated with a version of a function is named as follows:

- *location* is set to the value of the CURRENT SERVER special register
- *collection-id* (schema) for the package is the same as the schema qualifier of the function
- *package-id* is the same as the specific name of the function
- *version-id* is the same as the version identifier for the version of the function

The package is generated using the bind options that correspond to the implicitly or explicitly specified function options. In addition to the corresponding bind options, the package is generated using the following bind options:

- FLAG(I)
- SQLERROR(NOPACKAGE)
- ENABLE(*)

Considerations for SQL processor programs:

SQL processor programs (such as SPUFI, the command line processor, and DSNTEP2) might not correctly parse SQL statements in the routine body that are ended with semicolons. These processor programs accept multiple SQL statements as input when each statement is separated with a terminator character. Processor programs that use a semicolon as the SQL statement terminator might truncate an ALTER FUNCTION statement with embedded semicolons and pass only a portion of the statement to DB2. Therefore, you might need to change to SQL terminator character for these processor programs.

Correspondence of function options to bind command options:

The following table lists options for CREATE FUNCTION and ALTER FUNCTION and the corresponding bind command option. See BIND and REBIND options (DB2 Commands) for information about the BIND command options.

Table 95. Correspondence of function options to bind options

CREATE FUNCTION or ALTER FUNCTION option	**bind command option**
APPLICATION ENCODING SCHEME	ENCODING(ASCII), ENCODING(EBCDIC), ENCODING(UNICODE)
ARCHIVE SENSITIVE NO	ARCHIVESENSITIVE(NO)
ARCHIVE SENSITIVE YES	ARCHIVESENSITIVE(YES)
BUSINESS_TIME SENSITIVE NO	BUSTIMESENSITIVE(NO)
BUSINESS_TIME SENSITIVE YES	BUSTIMESENSITIVE(YES)
CURRENTDATA NO	CURRENTDATA(NO)
CURRENTDATA YES	CURRENTDATA(YES)

Table 95. Correspondence of function options to bind options (continued)

CREATE FUNCTION or ALTER FUNCTION option	bind command option
DYNAMICRULES	DYNAMICRULES(RUN), DYNAMICRULES(BIND), DYNAMICRULES(DEFINEBIND), DYNAMICRULES(DEFINERUN), DYNAMICRULES(INVOKEBIND), DYNAMICRULES(INVOKERUN)
ISOLATION LEVEL	ISOLATION(RR), ISOLATION(RS), ISOLATION(CS), ISOLATION(UR)
OPTHINT	OPTHINT
PACKAGE OWNER	OWNER
QUALIFIER	QUALIFIER
REOPT ALWAYS	REOPT(ALWAYS)
REOPT NONE	REOPT(NONE)
REOPT ONCE	REOPT(ONCE)
ROUNDING DEC_ROUND_CEILING	ROUNDING(CEILING)
ROUNDING DEC_ROUND_DOWN	ROUNDING(DOWN)
ROUNDING DEC_ROUND_FLOOR	ROUNDING(FLOOR)
ROUNDING DEC_ROUND_HALF_DOWN	ROUNDING(HALFDOWN)
ROUNDING DEC_ROUND_HALF_EVEN	ROUNDING(HALFEVEN)
ROUNDING DEC_ROUND_HALF_UP	ROUNDING(HALFUP)
ROUNDING DEC_ROUND_UP	ROUNDING(UP)
SQL PATH	PATH
SYSTEM_TIME SENSITIVE NO	SYSTIMESENSITIVE(NO)
SYSTEM_TIME SENSITIVE YES	SYSTIMESENSITIVE(YES)
VALIDATE BIND	VALIDATE(BIND)
VALIDATE RUN	VALIDATE(RUN)
WITH EXPLAIN	EXPLAIN(YES)
WITHOUT EXPLAIN	EXPLAIN(NO)
WITH IMMEDIATE WRITE	IMMEDWRITE(YES)
WITHOUT IMMEDIATE WRITE	IMMEDWRITE(NO)

Invalidation of packages:

When a version of an SQL function is altered to change any option that is specified for the active version, all packages that refer to that function are marked invalid. In addition, when certain attributes of an SQL function are changed, the body of the function might be rebound or regenerated. The following table summarizes when implicit rebind and regeneration occurs when specific options are changed. A value of 'Y' in a row indicates that a rebind or regeneration occurs if the option is changed for a version of the function. A value of 'N' in a row indicates that a rebind or regeneration does not occur.

Table 96. CREATE FUNCTION and ALTER FUNCTION options that result in rebind or regeneration of the function when changed

CREATE FUNCTION or ALTER FUNCTION option	**Change requires rebind of invoking application**	**Change results in implicit rebind of non-control statements in the body of the function**	**Change results in implicit regeneration of the entire body of the function**
ALLOW DEBUG MODE, DISALLOW DEBUG MODE, or DISABLE DEBUG MODE	Y **1** **2**	Y **1**	Y
APPLICATION ENCODING SCHEME	Y	Y	Y
ARCHIVE SENSITIVE	Y	Y	Y
ASUTIME	Y	N	N
BUSINESS_TIME SENSITIVE	Y	Y	Y
CURRENTDATA	N	Y	N
DATE FORMAT	Y	Y	Y
DECIMAL	Y	Y	Y
DYNAMICRULES	N	Y	N
FOR UPDATE CLAUSE OPTIONAL or FOR UPDATE CLAUSE REQUIRED	Y	Y	Y
INHERIT SPECIAL REGISTERS or DEFAULT SPECIAL REGISTERS	Y	N	N
ISOLATION LEVEL	N	Y	N
MODIFIES SQL DATA, READS SQL DATA, or CONTAINS SQL	Y	Y	Y
NOT DETERMINISTIC or DETERMINISTIC	N	N	N
OPTHINT	N	Y	N
PACKAGE OWNER	N	Y	N
QUALIFIER	N	Y	N
REOPT	N	Y	N
ROUNDING	Y	Y	Y
SQL PATH	N	Y	N
SYSTEM_TIME SENSITIVE	Y	Y	Y
TIME FORMAT	Y	Y	Y

Table 96. CREATE FUNCTION and ALTER FUNCTION options that result in rebind or regeneration of the function when changed (continued)

CREATE FUNCTION or ALTER FUNCTION option	Change requires rebind of invoking application	Change results in implicit rebind of non-control statements in the body of the function	Change results in implicit regeneration of the entire body of the function
VALIDATE RUN or VALIDATE BIND	N	Y	N
WITH EXPLAIN or WITHOUT EXPLAIN	N	Y	N
WITH IMMEDIATE WRITE or WITHOUT IMMEDIATE WRITE	N	Y	N
WLM ENVIRONMENT FOR DEBUG MODE	Y	N	N

Note:

1 The function package is rebound or regenerated if a value of ALLOW DEBUG MODE is changed to DISALLOW DEBUG MODE

2 Invoking applications are invalidated if a value of DISALLOW DEBUG MODE is changed to DISABLE DEBUG MODE

Considerations for the SYSENVIRONMENTS catalog table:

An ALTER statement that specifies new environment settings will result in a new row being added to the SYSENVIRONMENTS catalog table. The new row will be added even if an error is subsequently encountered during processing of the statement. Thus, a new SYSENVIRONMENT row might be added to the table even for an ALTER statement that fails.

Dependent objects:

An SQL routine is dependent on objects that are referenced in the routine body.

Altering a function from NOT SECURED to SECURED:

Typically, the security administrator will examine the data that is accessed by a function, ensure that it is secure, and grant the CREATE_SECURE_OBJECT privilege to the user that requires privileges to change the user-defined function to be secured. After the function is changed to SECURED, the security administrator will revoke the CREATE_SECURE_OBJECT privilege from the owner of the function.

The function is considered secure after the ALTER FUNCTION statement is executed. DB2 treats the SECURED attribute as an assertion that declares that the security administrator has established an audit procedure for all changes to the user-defined function. DB2 assumes that such a control audit procedure is in place for all subsequent ALTER FUNCTION statements or changes to external packages.

Packages and statements in the dynamic statement cache that reference the function are invalidated.

Altering a function from SECURED to NOT SECURED:

Packages and statements in the dynamic statement cache that reference the function are invalidated when the function is changed from SECURED to NOT SECURED. An function that is not secured might negatively impact performance if that function accesses data in a table that is using row

access control or column access control. To minimize the performance impact, either change the function to use the SECURED option or deactivate row access control or column access control for the table that the function is accessing.

Invoking other user-defined functions in a secure function:
When a secure user-defined function is referenced in an SQL data change statement that references a table that is using row access control or column access control, and if the secure user-defined function invokes other user-defined functions, the nested user-defined functions are not validated as secure. If those nested functions can access sensitive data, the security administrator needs to ensure that those functions are allowed to access sensitive data and should ensure that a change control audit procedure has been established for all changes to those functions.

The SECURE column in the DSN_FUNCTION_TABLE EXPLAIN table:
The SECURE column in the DSN_FUNCTION_TABLE EXPLAIN table indicates if a user-defined function is considered secure.

Deploying a non-inline SQL function:
When a BIND DEPLOY command is issued to deploy a non-inline SQL function to a target location, the SECURED and NOT SECURED options are included in the deployment process.

When deploying a non-inline SQL function, if a function with the same target name does not exist at the target location, the deployed function is created as a new function at the target location with the same SECURED or NOT SECURED option that is specified (or the default of NOT SECURED is used) in the source function of the deployment.

When deploying a non-inline SQL function, if a function with the same target name already exists at the target location, the deployed function is either added as a new version of the function or is used to replace an existing version of the function. The SECURED or NOT SECURED option of the deployed function must be the same as that of the existing function at the target location

Compatibilities:
For compatibility with the CREATE FUNCTION (SQL scalar) statement, the following clause can be specified, but will be ignored:

- LANGUAGE SQL

Optional syntax:
To provide compatibility with the syntax of the CREATE FUNCTION statement, the following options can also be specified:

- SPECIFIC
- PARAMETER CCSID

However, if these options are specified, the value for the option must be the same as the value that is already in effect for the function.

Alternative syntax and synonyms:
To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords:

- VARIANT as a synonym for **NOT DETERMINISTIC**
- NOT VARIANT as a synonym for **DETERMINISTIC**
- NULL CALL as a synonym for **CALLED ON NULL INPUT**
- NOT NULL CALL as a synonym for **RETURNS NULL ON NULL INPUT**
- TIMEZONE can be specified as an alternative to TIME ZONE

For an inline SQL scalar function, the RETURNS clause and the clauses in the *option-list* can be specified in any order. For a non-inline SQL scalar function, the RETURNS clause must precede the *options-list*. For both inline and non-inline SQL scalar functions, the *RETURN-statement* must be specified after the RETURNS clause and the *options-list* in the routine body.

Examples

Example 1: Modify the definition for an SQL function to indicate that the function is deterministic.

```
 ALTER FUNCTION MY_UDF1
DETERMINISTIC;
```

Example 2: The following statement changes the existing function options for the active version of the REVERSE SQL function. If you need to change a different version of the function, you would specify **VERSION** *routine-version-id* in place of **ACTIVE VERSION**. Note, the **ALTER** clause that precedes the version specification can be omitted:

```
 ALTER FUNCTION REVERSE
ALTER ACTIVE VERSION
NOT DETERMINISTIC
ALLOW DEBUG MODE;
```

Example 3: To change the function body of any existing version of a function, you need to use the **REPLACE** clause. The following statement changes both the function body and the existing SQL data access option for the version V2 of the REVERSE function. The list of parameters is specified even though no changes are made to the list. To replace an existing version of the function, you must specify the list of parameters, **RETURNS** clause, any options that are to have non-default values (even if those options are already specified in the version of the function that you are replacing), and the body of the function, as in the following statement:

```
ALTER FUNCTION REVERSE(INSTR VARCHAR(4000))
REPLACE VERSION V2 (INSTR VARCHAR(4000))
RETURNS VARCHAR(4000)
DETERMINISTIC
NO EXTERNAL ACTION
CONTAINS SQL
 BEGIN
 DECLARE REVSTR, RESTSTR VARCHAR(4000) DEFAULT '';
 DECLARE LEN INT;
 IF INSTR IS NULL THEN
  RETURN NULL;
 END IF;
 SET RESTSTR = INSTR;
 SET LEN = LENGTH(INSTR);
 WHILE LEN > 0 DO
  SET (REVSTR, RESTSTR, LEN) = (SUBSTR(RESTSTR, 1, 1) CONCAT
    REVSTR, SUBSTR(RESTSTR, 2, LEN - 1), LEN - 1);
 END WHILE;
 RETURN REVSTR;
  END
```

Example 4: To add a new version of an existing function, use the ADD VERSION clause. The following statement adds a new version of the REVERSE function to combine two SET statements into one SET statement. The list of parameters is specified even though the new version of the function uses the same parameters as the existing version of the function. To add a new version of the function, you must specify the list of parameters, RETURNS clause, any options that will have

non-default values, and the body of the function, as in the following statement, which creates version V3 of the REVERSE function:

```
ALTER FUNCTION REVERSE(INSTR VARCHAR(4000))
ADD VERSION V3 (INSTR VARCHAR(4000))
RETURNS VARCHAR(4000)
DETERMINISTIC
NO EXTERNAL ACTION
CONTAINS SQL
 BEGIN
 DECLARE REVSTR, RESTSTR VARCHAR(4000) DEFAULT '';
 DECLARE LEN INT;
 IF INSTR IS NULL THEN
  RETURN NULL;
 END IF;
 SET (RESRSTR, LEN) = (INSTR, LENGTH(INSTR));
 WHILE LEN > 0 DO
  SET (REVSTR, RESTSTR, LEN) = (SUBSTR(RESTSTR, 1, 1) CONCAT
    REVSTR, SUBSTR(RESTSTR, 2, LEN - 1), LEN - 1);
 END WHILE;
 RETURN REVSTR;
  END
```

Example 5: To change the currently active version of the function, you must specify the ACTIVATE VERSION clause on the ALTER FUNCTION statement, even if the version you want to be the active version has just been defined. The following statement causes version V3 of the REVERSE SQL function to be the currently active version:

```
ALTER FUNCTION REVERSE(INSTR VARCHAR(4000))
ACTIVATE VERSION V3;
```

Example 6: To regenerate the currently active version of the function, you must specify the REGENERATE clause, as in the following statement:

```
ALTER FUNCTION REVERSE(INSTR VARCHAR(4000))
REGENERATE ACTIVE VERSION;
```

ALTER FUNCTION (SQL table)

The ALTER FUNCTION (SQL table) statement changes the description of a user-defined SQL table function at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set that is defined below must include at least one of the following privileges or authorities:

- Ownership of the function
- The ALTERIN privilege on the schema
- SYSADM authority
- SYSCTRL authority
- System DBADM

The authorization ID that matches the schema name implicitly has the ALTERIN privilege on the schema.

If the authorization ID that is used to alter the function has installation SYSADM authority, the function is identified as system-defined function when the function definition is reevaluated.

If a distinct type is referenced (i.e. as the data type of an SQL variable in the body of the function), the privilege set must also include at least one of the following:

- Ownership of the distinct type
- The USAGE privilege on the distinct type
- SYSADM authority

At least one of the following privileges is required if the SECURED option is specified or if the function is currently secured and the NOT SECURED option is specified:

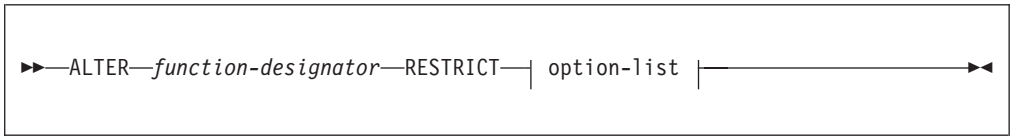
- SECADM authority
- CREATE_SECURE_OBJECT privilege

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the package.

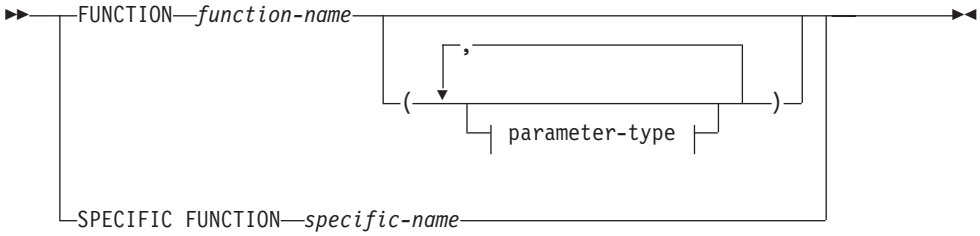
If the statement is dynamically prepared, the privilege set is the set of privileges that are held by the SQL authorization IDs of the process. The specified routine name can include a schema name (a qualifier). However, if the schema name is not the same as one of these SQL authorization IDs, one of the following conditions must be met:

- The privilege set includes SYSADM authority
- The privilege set includes SYSCTRL authority
- The SQL authorization ID of the process has the ALTERIN privilege on the schema

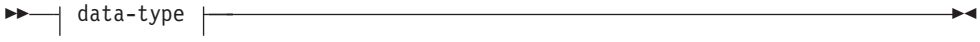
Syntax



function-designator:

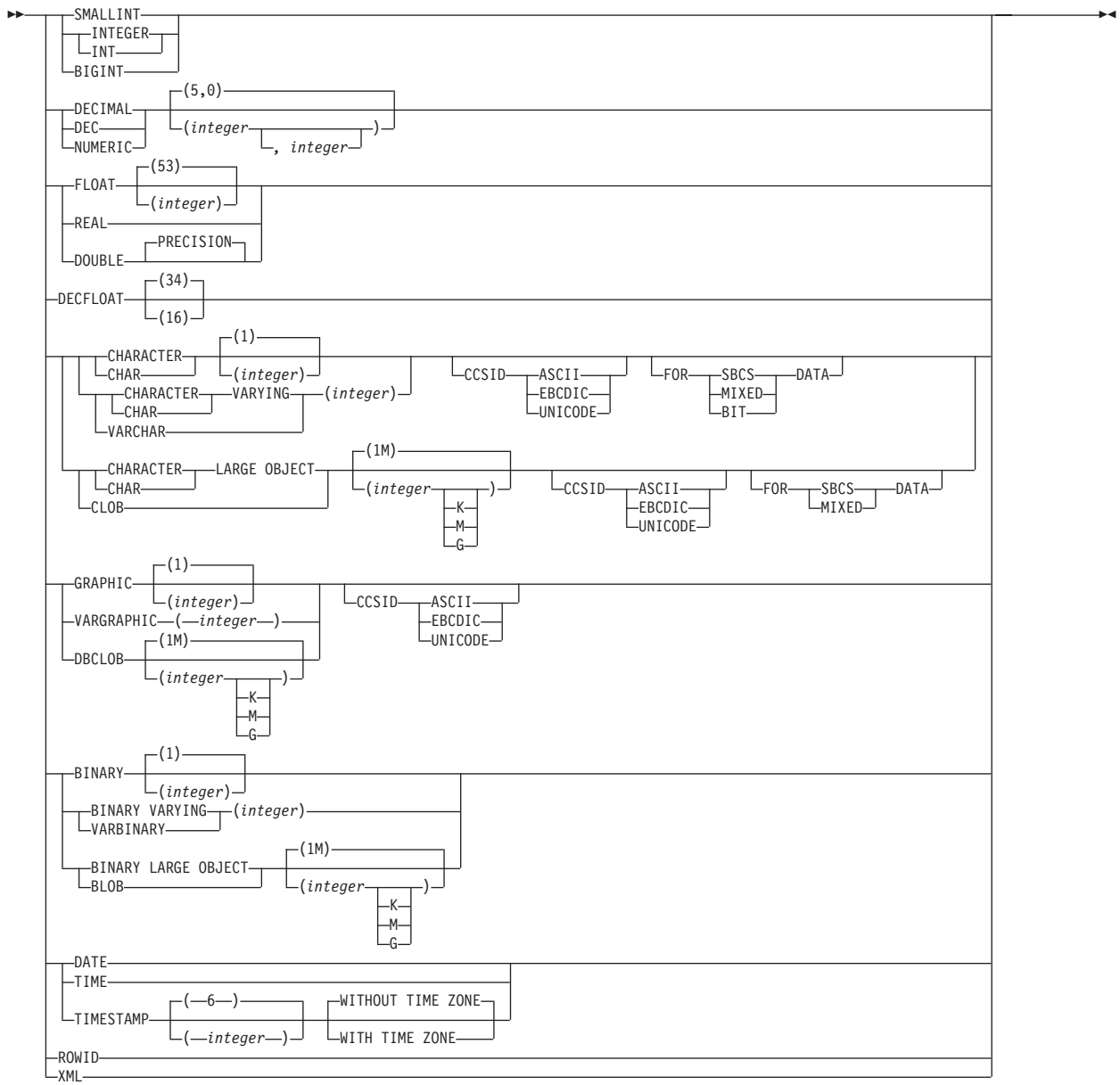


parameter-type:

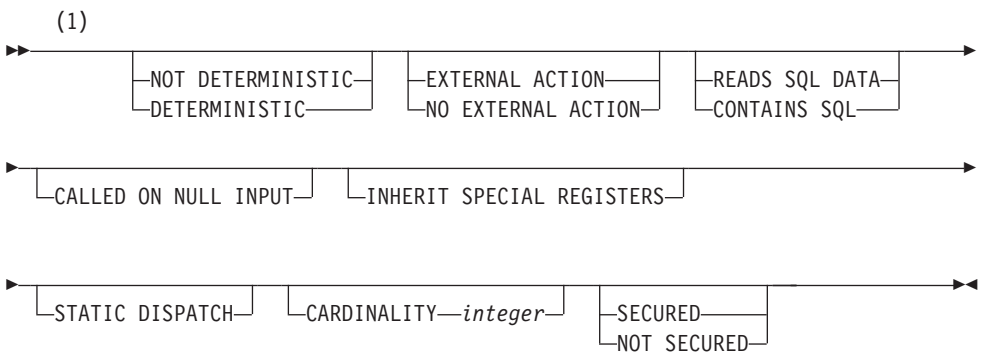


data-type:

built-in-type:



option-list:



Notes:

1 The options in the *option-list* can be specified in any order. However, the same clause cannot be specified more than one time.

Description

FUNCTION *function-name*

Identifies the SQL table function by its function name. The identified function must be an SQL table function.

There must be exactly one function with *function-name* in the schema. The function can have any number of input parameters. If the schema does not contain a function with *function-name*, or contains more than one function with this name, an error is returned.

FUNCTION *function-name* **(***parameter-type***, ...)**

Identifies the SQL function by its function signature, which uniquely identifies the function.

A function with the function signature must exist in the explicitly or implicitly specified schema.

function-name

Identifies the function name of the SQL function. If the function was defined with a table parameter (the **LIKE TABLE** *name* **AS LOCATOR** clause was specified in the CREATE FUNCTION statement to indicate that one of the input parameters is a transition table), the function signature cannot be used to uniquely identify the function. Instead, use one of the other syntax variations to identify the function with its function name, if unique, or its specified parameters.

If *function-name*() is specified, the function that is identified must have zero parameters.

parameter-type

Identifies the number of parameters of the function.

data-type

Identifies the data type of each input parameter of the function. The data type of each parameter must match the data type that was specified in the CREATE FUNCTION statement for the parameter in the corresponding position. The number of data types and the logical concatenation of the

data types are used to uniquely identify the function. Therefore, you cannot change the number of parameters or the data types of the parameters.

For data types that have a length, precision, or scale attribute, you can use a set of empty parentheses, specify a value, or accept the default values:

- Empty parentheses indicate that DB2 is to ignore the attribute when determining whether the data types match.

  For example, DEC() will be considered a match for a parameter of a function defined with a data type of DEC(7,2). Similarly DECFLOAT() will be considered a match for DECFLOAT(16) or DECFLOAT(34).

  FLOAT cannot be specified with empty parentheses because its parameter value indicates different data types (REAL or DOUBLE).
- If you use a specific value for a length, precision, or scale attribute, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

  The specific value for FLOAT(*n*) does not have to exactly match the defined value of the source function because `1<=n<= 21` indicates REAL and `22<=n<=53` indicates DOUBLE. Matching is based on whether the data type is REAL or DOUBLE.
- If length, precision, or scale is not explicitly specified and empty parentheses are not specified, the default length of the data type is implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

For data types with a subtype or encoding scheme attribute, specifying the **FOR** *subtype* **DATA** clause or the **CCSID** clause is optional. Omission of either clause indicates that DB2 is to ignore the attribute when determining whether the data types match. If you specify either clause, it must match the value that was implicitly or explicitly specified in the CREATE FUNCTION statement.

See "CREATE FUNCTION" on page 1165 for more information on the specification of the parameter list.

RESTRICT
Indicates that the function will not be altered or replaced it if is referenced by any function, materialized query table, procedure, trigger, or view.

NOT DETERMINISTIC or DETERMINISTIC
Specifies whether the function returns the same results each time that the function is invoked with the same input arguments. DB2 does not verify that the function program is consistent with the specification of NOT DETERMINISTIC or DETERMINISTIC.

NOT DETERMINISTIC
Specifies that the function might not return the same result table each time that the function is invoked with the same input arguments, even when the referenced data in the database has not changed. The function depends on some state values that might affect the results. DB2 uses this information to disable the merging of views and table expressions when processing SELECT and SQL data change statements that refer to this function. An example of a table function that is not deterministic is one which references special registers, other functions that are not deterministic, or a sequence in a way that affects the table function's result table.

DETERMINISTIC
Specifies that the function always returns the same result table each time that the function is invoked with the same input arguments (provided that the referenced data in the database has not changed). DB2 uses this information to enable the merging of views and table expressions for SELECT and SQL data change statements that refer to this function.

If applicable, specify DETERMINISTIC to prevent non-optimal access paths from being chosen for SQL statements that refer to this function.

EXTERNAL ACTION or NO EXTERNAL ACTION
Specifies whether the function contains an external action. DB2 does not verify that the function program is consistent with the specification of EXTERNAL ACTION or NO EXTERNAL ACTION.

EXTERNAL ACTION
The function performs some external action (outside the scope of the function program). Thus, the function must be invoked with each successive function invocation. EXTERNAL ACTION must be specified if the function invokes another function that has external actions.

NO EXTERNAL ACTION
The function does not perform any external action. It need not be called with each successive function invocation. Functions that are defined with NO EXTERNAL ACTION might perform better than functions that are defined with EXTERNAL ACTION because the function might not be invoked for each successive function invocation.

READS SQL DATA or CONTAINS SQL
Specifies the classification of SQL statements that the function (any routine that is invoked from this function) can execute. DB2 verifies that the SQL statements that the function issues are consistent with this specification.

READS SQL DATA
Specifies that the function can execute statements with a data access indication of READS SQL DATA or CONTAINS SQL. The function cannot execute SQL statements that modify data.

CONTAINS SQL
Specifies that the function can execute only SQL statements with a data access indication of CONTAINS SQL. The function cannot execute statements that read or modify data.

CALLED ON NULL INPUT
Specifies that the function is called regardless of whether any of the input argument values are null, making the function responsible for testing for null argument values. The function might return an empty table, depending on the logic in the body of the function.

INHERIT SPECIAL REGISTERS
Specifies that existing values of special registers are inherited upon entry to the function.

STATIC DISPATCH
Specifies that at function resolution time, DB2 chooses a function based on the static (or declared) types of the function parameters.

CARDINALITY *integer*
Specifies an estimate of the expected number of rows that the function returns. The number is used for optimization purposes. The value of *integer* must be between 0 and 2147483647.

If a function has an infinite cardinality (the function never returns the end-of-table condition and always returns a row), a query that requires the end-of-table condition to work correctly will need to be interrupted.

SECURED or NOT SECURED
Specifies whether the function is considered secure.

SECURED
Specifies that the function is considered secure.

NOT SECURED
Specifies that the function is considered not secure. NOT SECURED must not be specified when a row permission or a column mask depends on the function.

When the function is invoked, the arguments of the function must not reference a column for which a column mask is enabled when the table is using active column access control.

Notes

Invalidation of packages:
When an SQL function is changed, all the packages that refer to that function are marked invalid.

Dependent objects:
An SQL routine is dependent on objects that are referenced in the routine body.

Altering a function from NOT SECURED to SECURED:
Typically, the security administrator will examine the data that is accessed by a function, ensure that it is secure, and grant the CREATE_SECURE_OBJECT privilege to the user that requires privileges to change the user-defined function to be secured. After the function is changed to SECURED, the security administrator will revoke the CREATE_SECURE_OBJECT privilege from the owner of the function.

The function is considered secure after the ALTER FUNCTION statement is executed. DB2 treats the SECURED attribute as an assertion that declares that the security administrator has established an audit procedure for all changes to the user-defined function. DB2 assumes that such a control audit procedure is in place for all subsequent ALTER FUNCTION statements or changes to external packages.

Packages and statements in the dynamic statement cache that reference the function are invalidated.

Altering a function from SECURED to NOT SECURED:
Packages and statements in the dynamic statement cache that reference the function are invalidated when the function is changed from SECURED to NOT SECURED. An function that is not secured might negatively impact performance if that function accesses data in a table that is using row access control or column access control. To minimize the performance impact, either change the function to use the SECURED option or deactivate row access control or column access control for the table that the function is accessing.

Invoking other user-defined functions in a secure function:
When a secure user-defined function is referenced in an SQL data change statement that references a table that is using row access control or column access control, and if the secure user-defined function invokes other

user-defined functions, the nested user-defined functions are not validated as secure. If those nested functions can access sensitive data, the security administrator needs to ensure that those functions are allowed to access sensitive data and should ensure that a change control audit procedure has been established for all changes to those functions.

The SECURE column in the DSN_FUNCTION_TABLE EXPLAIN table:
The SECURE column in the DSN_FUNCTION_TABLE EXPLAIN table indicates if a user-defined function is considered secure.

Compatibilities:
For compatibility with the CREATE FUNCTION (SQL table) statement, the following clause can be specified, but will be ignored:

- LANGUAGE SQL

Alternative syntax and synonyms:
To provide compatibility with previously releases of DB2 or other products in the DB2 family, DB2 supports the following keywords:

- VARIANT as a synonym for NOT DETERMINISTIC
- NOT VARIANT as a synonym for DETERMINISTIC
- NULL CALL as a synonym for CALLED ON NULL INPUT

Examples

Example 1: The following statement modifies the definition of an SQL table function to set the estimated cardinality to 10,000.

```
ALTER FUNCTION GET_TABLE
 RESTRICT CARDINALITY 10000;
```

ALTER INDEX

The ALTER INDEX statement changes the description of an index at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set that is defined below must include one of the following:

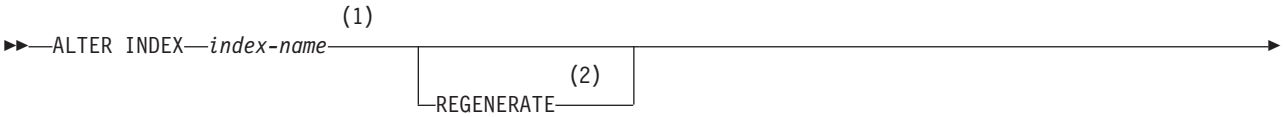
- Ownership of the index
- Ownership of the table on which the index is defined
- DBADM authority for the database that contains the table
- SYSADM or SYSCTRL authority
- System DBADM

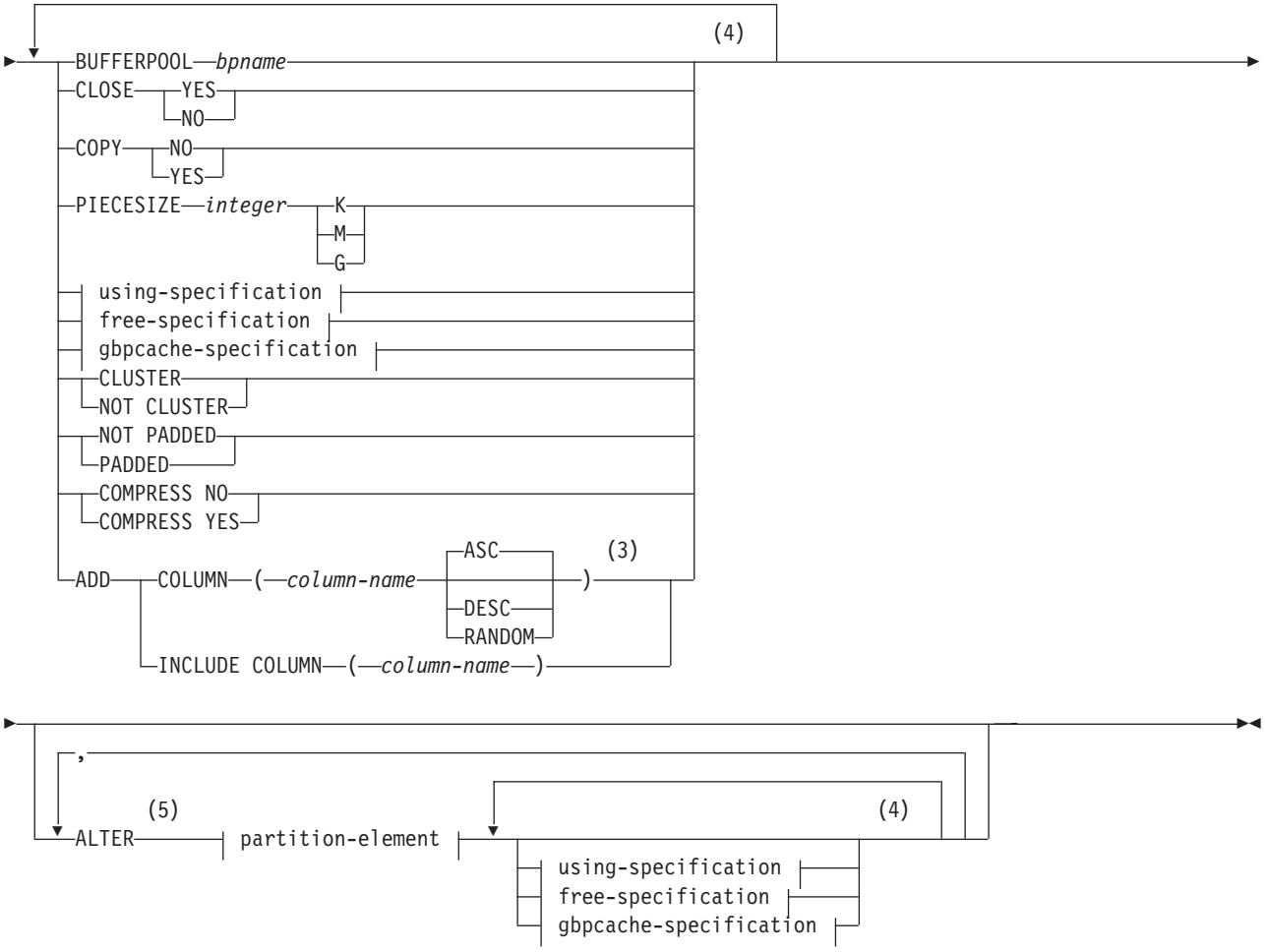
If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

If **BUFFERPOOL** or **USING STOGROUP** is specified, additional privileges could be needed, as explained in the description of those clauses.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the package. If the statement is dynamically prepared, the privilege set is the union of the privilege sets that are held by each authorization ID and role of the process.

Syntax





Notes:

1 At least one clause must be specified after *index-name*. It can be from the optional list or it can be **ALTER PARTITION**.

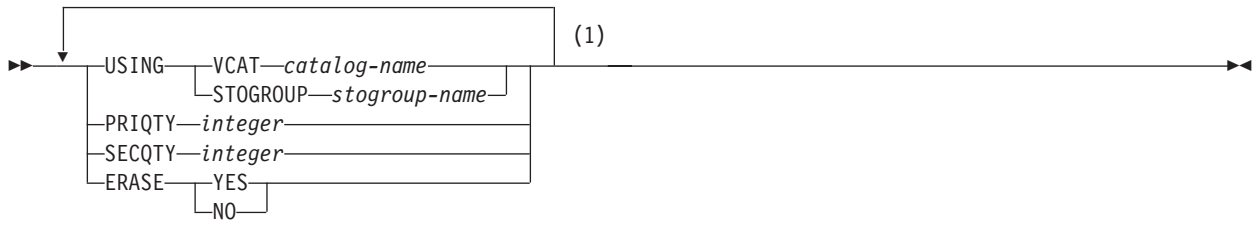
2 If **REGENERATE** is specified, it must be the only clause specified on the ALTER INDEX statement.

3 If **ADD COLUMN** and **PADDED** or **NOT PADDED** are specified, **ADD COLUMN** must be specified before **PADDED** or **NOT PADDED**.

4 The same clause must not be specified more than one time.

5 The **ALTER** clause can only be specified for partitioned indexes. The **ALTER** clause must be specified last.

using-specification:



Notes:

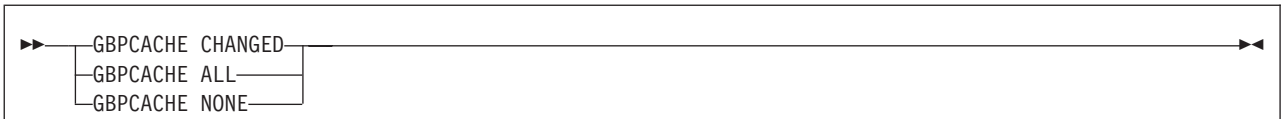
1 The same clause must not be specified more than one time.

free-specification:

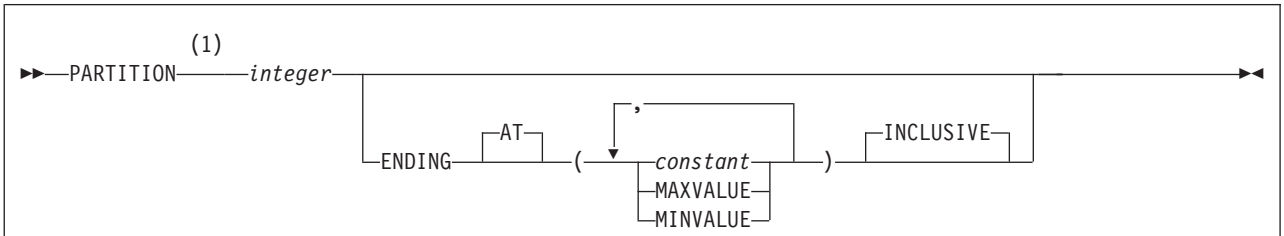
Notes:

1 The same clause must not be specified more than one time.

gbpcache-specification:



partition-element:



Notes:

1 If **PARTITION** is specified, either the **ENDING** clause, *using-specification*, *free-specification*, or *gbpcache-specification* should also be specified.

Description

index-name
Identifies the index to be changed or regenerated. The name must identify a

user-created index that exists at the current server. The name must not identify an index that is defined on a declared temporary table.

REGENERATE
Specifies that the index will be regenerated. The structure that represents the index definition is regenerated. The index definition will be composed from the catalog. Existing authorities and dependencies, if any, are retained. The catalog is updated with the regenerated index definition. The index is put into rebuild-pending state, all packages that depend on the index are invalidated, and catalog entries for the index statistics are deleted.

If the index cannot be successfully regenerated, an error is returned. In this case, the index must be dropped and recreated.

BUFFERPOOL *bpname*
Identifies the buffer pool that is to be used for the index. *bpname* must identify an activated 4K, 8 KB, 16 KB, or 32 KB buffer pool, and the privilege set must include SYSADM authority, SYSCTRL authority, or the USE privilege for the buffer pool.

A buffer pool with a smaller size should be chosen for indexes with random insert patterns. A buffer pool with a larger size should be chosen for indexes with sequential insert patterns.

If the index is changed to use index compression (the **COMPRESS YES** clause), the buffer pool must be 8 KB, 16 KB, or 32 KB in size.

The change is a pending definition change if all of the following conditions are true:

- The data sets of the index are created
- The index is defined on one of the following:
  - A table that is in a universal table space
  - An XML table that is associated with a base table that is in a universal table space
  - An auxiliary table that is associated with a base table that is in a universal table space
- There are pending definition changes for the index, table, or table space
- The buffer pool is changed to a buffer pool with a different size

If any of the previous conditions are not true, the change is an immediate change.

If the change is an immediate change, the change to the description of the index takes effect the next time the data sets of the index space are opened. The data sets can be closed and reopened by a **STOP DATABASE** command to stop the index followed by a **START DATABASE** command to start the index.

If the buffer pool is changed to a buffer pool with a different page size, and the change is an immediate change, the index is placed into REBUILD-pending status.

If the change is a pending definition change, the change is not reflected in the current definition or data at the time of the alter. Instead, the index is placed in an advisory REORG-pending (AREOR) state. A subsequent reorganization of the entire index with an appropriate utility will materialize the changes and apply the pending definition changes to the catalog and data.

CLOSE
Specifies whether the data set is eligible to be closed when the index is not

being used and the limit on the number of open data sets is reached. The change to the close rule takes effect the next time the data sets of the index space are opened.

YES
Eligible for closing.

NO Not eligible for closing.

If DSMAX is reached and there are no CLOSE YES page sets to close, CLOSE NO page sets will be closed.

COPY
Indicates whether the COPY utility is allowed for the index.

NO Does not allow full image or concurrent copies or the use of the RECOVER utility on the index.

YES
Allows full image or concurrent copies and the use the RECOVER utility on the index. For data sharing, changing **COPY** to **YES** causes additional SCA (Shared Communications Area) storage to be used until the next full or incremental image copy is taken or until **COPY** is set back to **NO**.

PIECESIZE *integer*
Specifies the maximum addressability of each data set for a secondary index. The **PIECESIZE** clause can only be specified for secondary indexes.

Be aware that when you alter the PIECESIZE value, the index is placed into page set REBUILD-pending (PSRBD) status. The entire index space becomes inaccessible. You must run the REBUILD INDEX or the REORG TABLESPACE utility to remove that status.

The subsequent keyword **K**, **M**, or **G**, indicates the units of the value that is specified in *integer*.

K Indicates that the *integer* value is to be multiplied by 1024 to specify the maximum data set size in bytes. *integer* must be a power of two between 1 and 268435456.

M Indicates that the *integer* value is to be multiplied by 1048576 to specify the maximum data set size in bytes. *integer* must be a power of two between 1 and 262144.

G Indicates that the *integer* value is to be multiplied by 1073741824 to specify the maximum data set size in bytes. *integer* must be a power of two between 1 and 256.

Table 97 shows the valid values for data set size, which depend on the size of the table space.

Table 97. Valid values of PIECESIZE clause

K units	M units	G units	Size attribute of table space
256K			
512K			
1024K	1M		
2048K	2M		
4096K	4M		
8192K	8M		
16384K	16M		

Table 97. Valid values of PIECESIZE clause (continued)

K units	M units	G units	Size attribute of table space
32768K	32M		
65536K	64M		
131072K	128M		
262144K	256M		
524288K	512M		
1048576K	1024M	1G	
2097152K	2048M	2G	
4194304K	4096M	4G	LARGE, DSSIZE 4G (or greater)
8388608K	8192M	8G	DSSIZE 8G (or greater)
16777216K	16384M	16G	DSSIZE 16G (or greater)
33554432K	32768M	32G	DSSIZE 32G (or greater)
67108864K	65536M	64G	DSSIZE 64G (or greater)
134217728K	131072M	128G	DSSIZE 128G (or greater)
268435456K	262144M	256G	DSSIZE 256G

The data set size limit for partitioned table spaces with more than 256 partitions is 4096.

begin using-specification block

The components of the *using-specification* are discussed below, first for non-partitioned indexes and then for partitioned indexes.

USING (specification for nonpartitioned indexes)
For nonpartitioned indexes, the **USING** clause specifies whether the data sets for the index are to be managed by the user or managed by DB2. The **USING** clause applies to every data set that can be used for the index.

If you specify **USING**, the index must be in the stopped state when the ALTER INDEX statement is executed. See Altering storage attributes to determine how and when changes take effect.

VCAT *catalog-name*
Specifies a user-managed data set with a name that starts with the specified catalog name. You must specify the catalog name in the form of an SQL identifier. Thus, you must specify an alias if the name of the integrated catalog facility catalog is longer than eight characters. When the new description of the index is applied, the integrated catalog facility catalog must contain an entry for the data set the conforms to the DB2 naming conventions described in *DB2 Administration Guide*.

One or more DB2 subsystems could share integrated catalog facility catalogs with the current server. To avoid the chance of having one of those subsystems attempt to assign the same name to different data sets, select a value for *catalog-name* that is not used by the other DB2 subsystems.

STOGROUP *stogroup-name*
Specifies using a DB2-managed data set that resides on a volume of the specified storage group. *stogroup-name* must identify a storage group that exists at the current server and the privilege set must include SYSADM authority, SYSCTRL authority, or the USE privilege for the storage group.

When the new description of the index is applied, the description of the storage group must include at least one volume serial number. Each volume serial number must identify a volume that is accessible to z/OS for dynamic allocation of the data set, and all identified volumes must be of the same device type. Furthermore, the integrated catalog facility catalog used for the storage group must not contain an entry for the data set.

If you specify **USING STOGROUP** and the current data set is DB2-managed, omission of the **PRIQTY**, **SECQTY**, or **ERASE** clause is an implicit specification of the current value of the omitted clause.

If you specify **USING STOGROUP** to convert from user-managed data sets to DB2-managed data sets:

- Omission of the **PRIQTY** clause is an implicit specification of the default value. For information on how DB2 determines the default value, see Rules for primary and secondary space allocation.
- Omission of the **SECQTY** clause is an implicit specification of the default value. For information on how DB2 determines the default value, see Rules for primary and secondary space allocation.
- Omission of the **ERASE** clause is an implicit specification of **ERASE NO**.

PRIQTY *integer*

Specifies the minimum primary space allocation for a DB2-managed data set. This clause can be specified only if the data set is currently managed by DB2 and USING VCAT is not specified.

If **PRIQTY** is specified (with a value other than -1), the primary space allocation is at least *n* kilobytes, where *n* is:

12 If *integer* is less than 12

integer
If *integer* is between 12 and 4194304

2097152
If both of the following conditions are true:
- *integer* is greater than 2097152.
- The index is a non-partitioned index on a table space that is not defined with the LARGE or DSSIZE attribute.

4194304
If *integer* is greater than 4194304

If **PRIQTY** -1 is specified, DB2 uses a default value for the primary space allocation. For information on how DB2 determines the default value for primary space allocation, see Rules for primary and secondary space allocation.

If **USING STOGROUP** is specified and **PRIQTY** is omitted, the value of **PRIQTY** is its current value. (However, if the current data set is being changed from being user-managed to DB2-managed, the value is its default value. See the description of **USING STOGROUP**.)

If you specify **PRIQTY** and do not specify a value of -1, DB2 specifies the primary space allocation to access method services using the smallest multiple of 4 KB not less than *n*. The allocated space can be greater than the amount of space requested by DB2. For example, it could be the smallest number of tracks that will accommodate the space requested. To more closely estimate the actual amount of storage, see DEFINE CLUSTER command (DFSMS Access Method Services for Catalogs).

When determining a suitable value for **PRIQTY**, be aware that two of the pages of the primary space could be used by DB2 for purposes other than storing index entries.

SECQTY *integer*
Specifies the minimum secondary space allocation for a DB2-managed data set. This clause can be specified only if the data set is currently managed by DB2 and **USING VCAT** is not specified.

If **SECQTY** -1 is specified, DB2 uses a default value for the secondary space allocation.

If **USING STOGROUP** is specified and **SECQTY** is omitted, the value of **SECQTY** is its current value. (However, if the current data set is being changed from being user-managed to DB2-managed, the value is its default value. See the description of **USING STOGROUP**.)

For information on the actual value that is used for secondary space allocation, whether you specify a value or DB2 uses a default value, see Rules for primary and secondary space allocation.

If you specify **SECQTY** and do not specify a value of -1, DB2 specifies the secondary space allocation to access method services using the smallest multiple of 4 KB not less than *n*. The allocated space can be greater than the amount of space requested by DB2. For example, it could be the smallest number of tracks that will accommodate the space requested. To more closely estimate the actual amount of storage, see DEFINE CLUSTER command (DFSMS Access Method Services for Catalogs).

ERASE
Indicates whether the DB2-managed data sets are to be erased when they are deleted during the execution of a utility or an SQL statement that drops the index.

NO Does not erase the data sets. Operations involving data set deletion will perform better than **ERASE YES**. However, the data is still accessible, though not through DB2.

YES
Erases the data sets. As a security measure, DB2 overwrites all data in the data sets with zeros before they are deleted.

This clause can be specified only if the data set is currently managed by DB2 and **USING VCAT** is not specified. If you specify **ERASE**, the index must be in the stopped state when the ALTER INDEX statement is executed. See Altering storage attributes to determine how and when changes take effect.

USING (specification for partitioned indexes:)
For a partitioned index, there is an optional **PARTITION** clause for each partition. A *using-specification* can be specified at the global level or at the partition level. A *using-specification* within a **PARTITION** clause applies only to that partition. A *using-specification* specified before any **PARTITION** clauses applies to every partition except those with a **PARTITION** clause with a *using-specification*.

For DB2-managed data sets, the values of **PRIQTY**, **SECQTY**, and **ERASE** for each partition are given by the first of these choices that applies:

- The values of **PRIQTY**, **SECQTY**, and **ERASE** given in the *using-specification* within the **PARTITION** clause for the partition. Do not use more than one *using-specification* in any **PARTITION** clause.

- The values of **PRIQTY**, **SECQTY**, and **ERASE** given in the *using-specification* before any **PARTITION** clause
- The current values of **PRIQTY**, **SECQTY**, and **ERASE**

For data sets that are being changed from user-managed to DB2-managed, the values of **PRIQTY**, **SECQTY**, and **ERASE** for each partition are given by the first of these choices that applies:

- The values of **PRIQTY**, **SECQTY**, and **ERASE** given in the *using-specification* within the PARTITION clause for the partition. Do not use more than one *using-specification* in any **PARTITION** clause.
- The values of **PRIQTY**, **SECQTY**, and **ERASE** given in a *using-specification* before any **PARTITION** clauses
- The default values of **PRIQTY**, **SECQTY**, and **ERASE**, which are:
  - **PRIQTY** 12
  - **SECQTY** 12, if **PRIQTY** is not specified in either *using-specification*, or 10% of **PRIQTY** or 3 times the index page size (whichever is larger) when **PRIQTY** is specified
  - **ERASE NO**

Any partition for which **USING** or **ERASE** is specified (either explicitly at the partition level or implicitly at the global level) must be in the stopped state when the ALTER INDEX statement is executed. See Altering storage attributes to determine how and when changes take effect.

VCAT *catalog-name*
Specifies a user-managed data set with a name that starts with the specified catalog name. You must specify the catalog name in the form of an SQL identifier. Thus, you must specify an alias if the name of the integrated catalog facility catalog is longer than eight characters.

If *n* is the number of the partition, the identified integrated catalog facility catalog must already contain an entry for the vth data set of the index, conforming to the DB2 naming convention for data sets described in *DB2 Administration Guide*.

One or more DB2 subsystems could share integrated catalog facility catalogs with the current server. To avoid the chance of having one of those subsystems attempt to assign the same name to different data sets, select a value for *catalog-name* that is not used by the other DB2 subsystems.

DB2 assumes one and only one data set for each partition.

STOGROUP *stogroup-name*
If **USING STOGROUP** is used, *stogroup-name* must identify a storage group that exists at the current server and the privilege set must include SYSADM authority, SYSCTRL authority, or the USE privilege for the storage group.

DB2 assumes one and only one data set for each partition.

For information on the **PRIQTY**, **SECQTY**, and **ERASE** clauses, see the description of those clauses in the *using-specification* for secondary indexes.

end using-specification block

begin free-specification block

FREEPAGE *integer*
Specifies how often to leave a page of free space when index entries are created as the result of executing a DB2 utility. One free page is left for every *integer* pages. The value of *integer* can range from 0 to 255. The change to the description of the index or partition has no effect until it is loaded or reorganized using a DB2 utility. Do not specify **FREEPAGE** for an implicitly created XML index.

PCTFREE *integer*
Determines the percentage of free space to leave in each nonleaf page and leaf page when entries are added to the index or partition as the result of executing a DB2 utility. The first entry in a page is loaded without restriction. When additional entries are placed in a nonleaf or leaf page, the percentage of free space is at least as great as *integer*.

The value of *integer* can range from 0 to 99, however, if a value greater than 10 is specified, only 10 percent of free space will be left in nonleaf pages. The change to the description of the index or partition has no effect until it is loaded or reorganized using a DB2 utility. Do not specify **PCTFREE** for an implicitly created XML index.

If the index is partitioned, the values of FREEPAGE and PCTFREE for a particular partition are given by the first of these choices that applies:

- The values of **FREEPAGE** and **PCTFREE** given in the **PARTITION** clause for that partition. Do not use more than one *free-specification* in any **PARTITION** clause.
- The values given in a *free-specification* before any **PARTITION** clauses.
- The current values of **FREEPAGE** and **PCTFREE** for that partition.

end free-specification block

begin gbpcache-specification block

GBPCACHE
Specifies what index pages are written to the group buffer pool in a data sharing environment. In a non-data-sharing environment, you can specify this option, but it is ignored.

CHANGED
When there is inter-DB2 read-write interest on the index or partition, updated pages are written to the group buffer pool. When there is no inter-DB2 read-write interest, the group buffer pool is not used. Inter-DB2 read-write interest exists when more than one member in the data sharing group has the index or partition open, and at least one member has it open for update.

If the index is in a group buffer pool that is defined as **GBPCACHE**(**NO**), **CHANGED** is ignored and no pages are cached to the group buffer pool.

ALL
Indicates that pages are to be cached to the group buffer pool as they are read in from DASD, with one exception. When the page set is not GBP-dependent and one DB2 data sharing member has exclusive read-write interest in that page set (no other group members have any interest in the page set), no pages are cached in the group buffer pool.

If the index is in a group buffer pool that is defined as **GBPCACHE**(**NO**), **ALL** is ignored and no pages are cached to the group buffer pool.

NONE
Indicates that no pages are to be cached to the group buffer pool. DB2 uses the group buffer pool only for cross-invalidation.

If you specify **NONE**, the index or partition must not be in group buffer pool recover-pending (GRECP) status.

If the index is partitioned, the value of **GBPCACHE** for a particular partition is given by the first of these choices that applies:

1. The value of **GBPCACHE** given in the **PARTITION** clause for that partition. Do not use more than one *gbpcache-specification* in any **PARTITION** clause.
2. The value given in a *gbpcache-specification* before any **PARTITION** clauses.
3. The current value of **GBPCACHE** for that partition.

If you specify **GBPCACHE** in a data sharing environment, the index or partition must be in the stopped state when the ALTER INDEX statement is executed. You cannot alter the GBPCACHE value for certain indexes on DB2 catalog tables; for more information, see "SQL statements allowed on the catalog" on page 2113.

end gbpcache-specification block

CLUSTER or NOT CLUSTER
Specifies whether the index is the clustering index for the table.

CLUSTER
The index is used as the clustering index for the table. This change takes effect immediately. Any subsequent insert operations will use the new clustering index. Existing data remains clustered by the previous clustering index until the table space is reorganized.

The implicit or explicit clustering index is ignored when data is inserted into a table space that is defined with **MEMBER CLUSTER**. Instead of using cluster order, DB2 chooses where to locate the data based on available space. The **MEMBER CLUSTER** attribute affects only data that is inserted with an insert operation; data is always loaded and reorganized in cluster order.

Do not specify **CLUSTER** in the following cases:

- The index is for an auxiliary table.
- **CLUSTER** was used already for a different index on the table.
- The index is an XML index.
- The index includes expressions.
- The index is for a table that uses hash organization.
- The index is the hash overflow index for a table.

NOT CLUSTER
The index is not used as the clustering index of the table. If the index is already defined as the clustering index, it continues to be used as the clustering index by DB2 and the REORG utility until clustering is explicitly changed by specifying **CLUSTER** for a different index.

Specifying **NOT CLUSTER** for an index that is not a clustering index is ignored.

If the index is the partitioning index for a table that uses index-controlled partitioning, the table is converted to use table-controlled partitioning. The

high limit key for the last partition is set to the highest possible value for ascending key columns or the lowest possible value for descending key columns.

COMPRESS NO or COMPRESS YES
Specifies whether the index data will be compressed. If the index is partitioned, this option will apply to all partitions.

When an index is changed from one compression option to another (either from COMPRESS YES to COMPRESS NO or COMPRESS NO to COMPRESS YES), the index marked as rebuild pending. For a non-partitioned index, the index will be placed in a page set rebuilding state. For a partitioned index, the index will be placed in rebuilding state.

COMPRESS NO
Specifies that index compression will be turned off.

COMPRESS YES
Specifies that the index will use index compression. **COMPRESS YES** can be specified for user-managed data sets only if the control interval size is 4K.

NOT PADDED or PADDED
Specifies how varying-length string columns are to be stored in the index. If the index contains no varying-length columns, this option is ignored, and a warning message is returned.

NOT PADDED
Specifies that varying-length string columns are not to be padded to their maximum length in the index. The length information for a varying-length column is stored with the key.

NOT PADDED is ignored and has no effect if the index is on an auxiliary table. Indexes on auxiliary tables are always padded.

When **PADDED** is changed to **NOT PADDED**, the maximum key length is recalculated with the varying-length formula (`2000 - n - 2m`, where *n* is the number of columns that can contain null values and *m* is the number of varying-length columns in the key). If it is possible that the index key length might exceed the maximum length (because when it was padded, the formula `2000 - n` was used), an error occurs.

PADDED
Specifies that varying-length string columns within the index are always padded with the default pad character to their maximum length.

When an index with at least one varying-length column is changed from **PADDED** to **NOT PADDED**, or vice versa, the index is placed in restricted rebuild-pending status (RBDP). The index cannot be accessed until it is rebuilt from the table (using the REBUILD INDEX, REORG TABLESPACE, or LOAD REPLACE utility). For nonpartitioned secondary indexes (NPSIs), the index is placed in page set rebuild-pending status (PSRBD), and the entire index must be rebuilt. In addition, packages that are dependent on the table are quiesced, and dynamically cached statements that are dependent on the index are invalidated.

Do not specify **PADDED** if the index is an XML index.

ADD COLUMN *column-name*
Adds *column-name* to the index. *column-name* must be unqualified, must identify a column of the table, must not be one of the existing columns of the

index, and must not be a LOB column, a DECFLOAT column, or a distinct-type column that is based on a LOB or DECFLOAT data type. The column cannot be a VARBINARY column or a distinct-type column that is based on a VARBINARY data type if the column is defined with the DESC attribute or if the index is defined with the PADDED attribute.

The column cannot be a timestamp with time zone column (or a column with a distinct type that is based on the timestamp with time zone data type) when the PARTITION or PARTITION BY RANGE clause is also specified.

The index must not already be defined with the BUSINESS_TIME WITHOUT OVERLAPS specification.

The total number of columns for the index cannot exceed 64.

If a column is added to an index that is defined with the EXCLUDE NULL KEYS clause, the index is placed in REBUILD-pending status.

If the index is defined with the EXCLUDE NULL KEYS clause, the specified column must allow null values.

For PADDED indexes, the sum of the length attributes of the columns must not be greater than `2000 - n`, where *n* is the number of columns that can contain null values. For NOT PADDED indexes, the sum of the length attributes of the columns must not be greater than `2000 - n -2m`, where *n* is the number of nullable columns and *m* is the number of varying-length columns.

The index cannot be any of the following types of indexes:

- A system-defined catalog index
- An index that enforces a primary key, unique key, or referential constraint, or matches a foreign key
- A partitioning index when index-controlled partitioning is being used
- A unique index required for a ROWID column defined as GENERATED BY DEFAULT
- An auxiliary index
- An XML index
- An index that includes expressions
- The hash overflow index for a table.

The index is put into rebuild-pending (RBDP) status in the following cases:

- *column-name* specifies is a ROWID column
- a column is added to a table, rows are inserted into the table, and the same column is added to an associated index all within the same commit scope
- a column is added to a table and then is added to an associated index in a separate commit scope

Otherwise, the index is put into an advisory reorg-pending (AREO*) state.

ASC
Index entries are put in ascending order by the column.

DESC
Index entries are put in descending order by the column.

RANDOM
Index entries are put in a random order by the column. **RANDOM** cannot be specified in the following cases:

- A varying length column is part of the index key and the index is defined with the NOT PADDED option.
- A column of the index key is defined as TIMESTAMP WITH TIME ZONE.
- The index is part of a partitioning key.

ADD INCLUDE (*column-name*)
Specifies an additional column to append to the set of index key columns of a unique index. Any column that is specified using **INCLUDE** *column-name*, is not used to enforce uniqueness. The included column might improve performance for some queries using index only access.

Columns that are specified in the ADD INCLUDE clause count towards the limits for the number of columns and the limits on the sum of the length attributes of the columns that are specified in the index. The total number of columns for the index cannot exceed 64.

column-name must be unqualified, must identify a column of the specified table, and must not be one of the existing columns of the index. *column-name* must not identify a LOB or DECFLOAT column (or a distinct type that is based on one of those types).

The INCLUDE clause cannot be specified for the following types of indexes:

- A system defined catalog index
- A non-unique index
- A partitioning index when index-controlled partitioning is used
- An auxiliary index
- An index on a foreign key
- An XML index
- An extended index
- An index that includes expressions
- An index that is created with the EXCLUDE NULL KEYS clause

If a column is added to both a table and an associated index within the same commit scope and the column is not a ROWID column, the index is placed in an advisory reorg-pending state (AREO*). Otherwise, the index is placed in a rebuild-pending state (RBDP).

ALTER PARTITION *integer*
Identifies the partition of the index to be altered. For an index that has *n* partitions, you must specify an integer in the range 1 to *n*. You must not use this clause under the following conditions:

- If the index is nonpartitioned
- If the index is defined on a table that contains an XML column and uses index-controlled partitioning

You must use this clause if the index is partitioned and you specify the **ENDING AT** clause.

ENDING AT(*constant***), MAXVALUE, or MINVALUE**
Specifies the highest value of the index key for the identified partition of the partitioning index. In this context, highest means highest in the sorting sequence of the index columns. In a column defined as ascending (ASC), highest and lowest have the usual meanings. In a column defined as descending (DESC), the lowest actual value is highest in the sorting sequence.

You must use at least one value (*constant*, **MAXVALUE**, or **MINVALUE**) after **ENDING AT** in each **PARTITION** clause. You can use as many values as there are columns in the key. The concatenation of all the values is the highest value of the key in the corresponding partition of the index. The length of each highest key value (also called the limit key) is the same as the length of the partitioning index

constant
Specifies a constant value with a data type that must conform to the rules for assigning that value to the column. If a string constant is longer or shorter than required by the length attribute of its column, the constant is either truncated or padded on the right to the required length. If the column is ascending, the padding character is X'FF'. If the column is descending, the padding character is X'00'. The precision and scale of a decimal constant must not be greater than the precision and scale of its corresponding column. A hexadecimal string constant (GX) cannot be specified.

MAXVALUE
Specifies a value greater than the maximum value for the limit key of a partition boundary (that is, all X'FF' regardless of whether the column is ascending or descending). If all of the columns in the partitioning key are ascending, a constant or the **MINVALUE** clause cannot be specified following **MAXVALUE**. After **MAXVALUE** is specified, all subsequent columns must be **MAXVALUE**.

MINVALUE
Specifies a value that is smaller than the minimum value for the limit key of a partition boundary (that is, all X'00' regardless of whether the column is ascending or descending). If all of the columns in the partitioning key are descending, a constant or the **MAXVALUE** clause cannot be specified following **MAXVALUE**. After **MINVALUE** is specified, all subsequent columns must be **MINVALUE**.

The key values are subject to the following rules:

- The first value corresponds to the first column of the key, the second value to the second column, and so on.
- If a key includes a ROWID column (or a column with a distinct type that is based on a ROWID data type), the values of the ROWID column are assumed to be in the range of X'000...00' to X'FFF...FF'. Only the first 17 bytes of the value that is specified for the corresponding ROWID column are considered.
- Using fewer values than there are columns in the key has the same effect as using the highest possible values for all omitted columns for an ascending index.
- If the key exceeds 255 bytes, only the first 255 bytes are considered.
- The highest value of the key in any partition must be lower than the highest value of the key in the next partition.
- The highest value of the key in the last partition depends on how the table space was defined. For table spaces created without the LARGE or DSSIZE option, the constants you specify after **ENDING AT** are not enforced. The highest value of the key that can be placed in the table is the highest possible value of the key.

  For table spaces created with the LARGE or DSSIZE options, the constants you specify after **ENDING AT** are enforced. The value specified for the last partition is the highest value of the key that can be

placed in the table. Any keys that are made invalid after the ALTER TABLE statement is executed are placed in a discard data set when you run the REORG utility. If the last partition is in reorg-pending status, regardless of whether you changed its limiting key values, you must specify a discard data set when you run the REORG utility.

ENDING AT must not be specified for any indexes defined on a table that uses table-controlled partitioning. Use ALTER TABLE ALTER PARTITION to modify the partitioning boundaries for a table that uses table-controlled partitioning.

INCLUSIVE
Specifies that the specified range values are included in the data partition.

Notes

Pending definition changes:
ALTER INDEX causes an immediate definition change to the specified index with the exception of ALTER INDEX BUFFERPOOL. ALTER INDEX BUFFERPOOL results in an immediate definition change except when all of the following conditions are true:

- The data sets of the index are created
- The index is defined on one of the following:
  - A table that is in a universal table space
  - An XML table that is associated with a base table this is in a universal table space
  - An auxiliary table that is associated with a base table that is in a universal table space
- There are pending definition changes for the index or the table space, or the buffer pool is changed to a buffer pool with a different size.

When ALTER INDEX causes a pending definition change, semantic validation and authorization checking are performed for the statement. However, the current definition of the index is not changed, and the index is placed in advisory REORG-pending (AREOR) state. If there are no pending definition changes for the table space, you can run the REORG INDEX utility with SHRLEVEL CHANGE or the REORG TABLESPACE utility with SHRLEVEL CHANGE or REFERENCE to enable the changes to the definition of the index. If pending definition changes also exist for the table space, you must run the REORG TABLESPACE utility with SHRLEVEL CHANGE or REFERENCE to enable the changes to the definition of the index (and the pending table space definition). When the pending definition changes are applied to the index, dependent packages might be invalidated.

Restrictions involving pending definition changes:
ALTER INDEX statements that result in a pending definition change are not allowed in the following cases:

- On the catalog, system objects, or objects in a workfile database
- If the definition of the table space is incomplete
- If the definition of the table on which the index is defined is incomplete
- If the ALTER INDEX statement also specifies options that will cause an immediate definition change

- If there are already pending definition changes to the index, ALTER INDEX to change from COMPRESS NO to COMPRESS YES is not allowed
- If there are already pending definition changes to the index or the table space that contains the index, the following are not allowed:
  - ALTER INDEX (with or without ALTER PARTITION) to change from a DB2-managed data set to a user-managed data set
  - ALTER INDEX to change the value of PIECESIZE
- If there are already pending definition changes to the containing table space or any objects within the table space, the following are not allowed:
  - ALTER INDEX REGENERATE to regenerate the index
  - ALTER INDEX ADD COLUMN to add a column to the index

Altering storage attributes:

The **USING**, **PRIQTY**, **SECQTY**, and **ERASE** clauses define the storage attributes of the index or partition. If you specify the **USING** or **ERASE** clause when altering storage attributes, the index or partition must be in the stopped state when the ALTER INDEX statement is executed. A **STOP DATABASE...SPACENAM...** command can be used to stop the index or partition.

If the catalog name changes, the changes take effect after you move the data and start the index or partition using the **START DATABASE...SPACENAM...** command. The catalog name can be implicitly or explicitly changed by the ALTER INDEX statement. The catalog name also changes when you move the data to a different device. See the procedures for moving data in *DB2 Administration Guide*.

Changes to the secondary space allocation (SECQTY) take effect the next time DB2 extends the data set; however, the new value is not reflected in the integrated catalog until you use the REORG, RECOVER, or LOAD REPLACE utility on the index or partition. Changes to the other storage attributes take effect the next time you use the REORG, RECOVER, or LOAD REPLACE utility on the index or partition. If you change the primary space allocation parameters or erase rule, you can have the changes take effect earlier if you move the data before you start the index or partition.

Altering indexes on DB2 catalog tables:

For details on altering options on catalog tables, see "SQL statements allowed on the catalog" on page 2113.

Size restriction for the object descriptor of an index in the SYSIBM.SYSOBDS catalog table:

The following case might result in an error being returned if the ALTER INDEX statement results in a versioned object descriptor that is larger than 30,000 bytes being added (or updated) in the SYSIBM.SYSOBDS catalog table:

- An ALTER INDEX statement that results in the first version of the object descriptor being generated for the index

You might need to drop and recreate the index if the object descriptor for the index exceeds 30,000 bytes.

Altering limit keys:

If you specify **ALTER PARTITION** *integer* **ENDING AT** to change the

limit key values of a partitioning index, the packages that are dependent on that index are marked invalid and go through automatic rebind the next time they are run.

Invalidation of packages:

When an index is altered, all the packages that refer to that index are marked invalid if one of the following conditions is true:

- A column is added to the index.
- The index is altered to be **PADDED** or **NOT PADDED**.
- The index is a partitioning index on a table that uses index-controlled partitioning, and one or more limit key values is altered.
- The index is altered to **REGENERATE**.

Restrictions on SQL data change statements in the same commit scope as ALTER INDEX:

SQL data change statements that affect an index cannot be performed in the same commit scope as ALTER INDEX statements that affect that index.

Altering indexes for tables that are involved in a clone relationship:

You cannot change any index for a table that is involved in a clone relationship (base table or clone table). If a change to an index is required, the clone table must be dropped, than the index can be changed. After the index is changed, the clone table can be created again.

Adding a varying length column to a key for a system with NOT PADDED as the default:

If the system default is NOT PADDED (the value of field PAD INDEXES BY DEFAULT on installation panel DSNTIPE is NO), no varying length columns are in the key, and the PADDED or NOT PADDED option is not explicitly specified when the index is created, the PADDED column of the SYSIBM.SYSINDEXES catalog table is populated with a blank value. If a varying length column is later added to the key, the value of the PADDED column in SYSIBM.SYSINDEXES is changed to 'Y' to indicate that the index is now a PADDED index.

Running utilities:

You cannot execute the ALTER INDEX statement while a DB2 utility has control of the index or its associated table space.

Alternative syntax and synonyms:

To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords when altering the partitions of a partitioned index:

- PART can be specified as a synonym for **PARTITION**. In addition, the **ALTER** keyword that precedes **PARTITION** is optional. In addition, if you alter more than one partition, specifying a comma between each **ALTER PARTITION** *integer* clause is optional.
- VALUES can be specified as a synonym for **ENDING AT**.

Although these keywords are supported as alternatives, they are not the preferred syntax.

Examples

Example 1: Alter the index DSN8B10.XEMP1. Indicate that DB2 is not to close the data sets that support the index when there are no current users of the index.

```
ALTER INDEX DSN8B10.XEMP1
  CLOSE NO;
```

Example 2: Alter the index DSN8B10.XPROJ1. Use BP1 as the buffer pool that is to be associated with the index, indicate that full image or concurrent copies on the index are allowed, and change the maximum size of each data set to 8 megabytes.

```
ALTER INDEX DSN8B10.XPROJ1
  BUFFERPOOL BP1
  COPY YES
  PIECESIZE 8M;
```

Example 3: Assume that index X1 contains a least one varying-length column and is a padded index. Alter the index to an index that is not padded.

```
ALTER INDEX X1
  NOT PADDED;
```

The index is placed in restricted rebuild-pending status (RBDP) and cannot be accessed until it is rebuilt from the table

Example 4: Alter partitioned index DSN8B10.DEPT1. For partition 3, leave one page of free space for every 13 pages and 13 percent of free space per page. For partition 5, leave one page for every 25 pages and 25 percent of free space. For all the other partitions, leave one page of free space for every 6 pages and 11 percent of free space. Ensure that index pages are cached to the group buffer pool for all partitions except partition 4. For partition 4, write pages only when there is inter-DB2 read-write interest on the partition.

```
ALTER INDEX DSN8B10.XDEPT1
   BUFFERPOOL BP1
   CLOSE YES
   COPY YES
   USING VCAT CATLGG
   FREEPAGE 6
   PCTFREE 11
   GBPCACHE ALL
   ALTER PARTITION 3
      USING VCAT CATLGG
      FREEPAGE 13
      PCTFREE 13,
   ALTER PARTITION 4
      USING VCAT CATLGG
      GBPCACHE CHANGED,
   ALTER PARTITION 5
      USING VCAT CATLGG
      FREEPAGE 25
      PCTFREE 25;
```

ALTER MASK

The ALTER MASK statement changes a column mask that exists at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

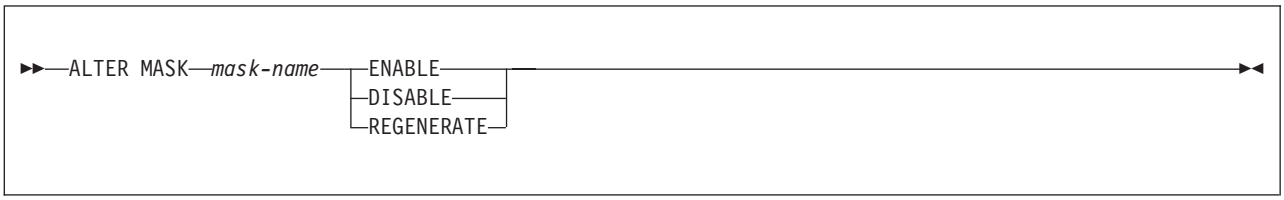
Authorization

The privilege set that is defined below must include the following authority:

- SECADM authority

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the package. If the statement is dynamically prepared, the privilege set is the union of the privilege sets that are held by each authorization ID and role of the process.

Syntax



Description

mask-name
Identifies the column mask to be altered. The name must identify a mask that exists at the current server.

ENABLE
Specifies that the column mask is to be enabled for column access control. If column access control is not currently activated for the table, the column mask will become effective when column access control is activated for the table. If column access control is currently activated for the table, the column mask becomes effective immediately and all packages and dynamic cached statements that reference the table are invalidated.

A column mask with a regeneration error cannot be enabled. To clear the status of the column mask, the column mask must be dropped and re-created with a modified definition.

ENABLE is ignored if the column mask is already defined as enabled for column access control.

DISABLE
Specifies that the column mask is to be disabled for column access control. If column access control is not currently activated for the table, the column mask will remain ineffective when column access control is activated for the table. If column access control is currently activated for the table, the column mask

becomes ineffective immediately and all packages and dynamic cached statements that reference the table are invalidated.

DISABLE is ignored if the column mask is already defined as disabled for column access control.

REGENERATE
Specifies that the column mask is to be regenerated. The column mask definition in the catalog is used, and existing dependencies and authorization, if any, are retained. The column mask definition is reevaluated as if the column mask was being created. The user-defined functions that are referenced in the column mask definition must be resolved to the same secure UDFs as that were resolved during the column mask creation.

Notes

Applying DB2 maintenance:
When DB2 maintenance is applied that affects how a column mask is generated, the column mask might need to be regenerated to ensure the column mask is still valid.

If the column mask is regenerated successfully, the status of the column mask is set to a blank in the catalog table. If the column mask is enabled and column access control is currently activated for the table, all packages and dynamic cached statements that reference the table are invalidated.

If the column mask cannot be regenerated successfully, an error is returned. The regeneration status of the column mask is an error. If the column mask is enabled and column access control is currently activated for the table, all packages and dynamic cached statements that reference the table are marked invalidated. To clear the status of the column mask, the column mask must be dropped and re-created with a modified definition. Or the column mask can be disabled if not disabled yet. A disabled column mask becomes ineffective to a column access control enforced table.

When the table is referenced in a data manipulation statement, the statement returns an error if any enabled column mask has an regeneration error.

Examples

Example 1:
Enable column mask M1.

```
ALTER MASK M1 ENABLE;
```

Example 2:
Regenerate column mask M1.

```
ALTER MASK M1 REGENERATE;

COMMIT;
```

ALTER PERMISSION

The ALTER PERMISSION statement alters a row permission that exists at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

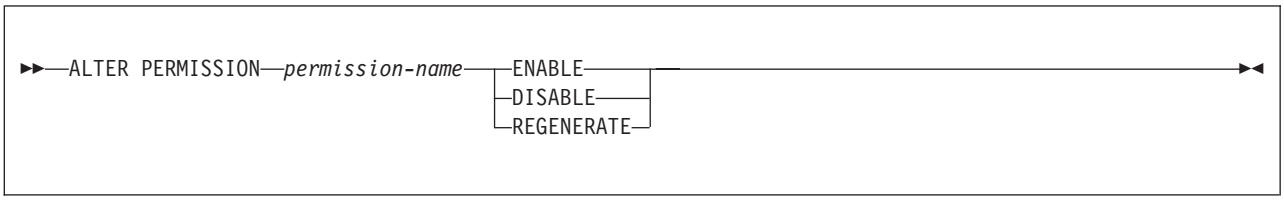
Authorization

The privilege set that is defined below must include the following authority:

SECADM authority

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the package. If the statement is dynamically prepared, the privilege set is the union of the privilege sets that are held by each authorization ID and role of the process.

Syntax



Description

permission-name

Identifies the permission to be altered. The name must identify a row permission that exists at the current server. The name must not identify a default row permission that is created implicitly by DB2.

ENABLE

Specifies that the row permission is to be enabled for row access control. If row access control is not currently activated for the table, the row permission will become effective when row access control is activated for the table. If row access control is currently activated for the table, the row permission becomes effective immediately and all packages and dynamic cached statements that reference the table are invalidated.

A row permission with a regeneration error cannot be enabled. To clear the status of the row permission, the row permission must be dropped and re-created with a modified definition.

ENABLE is ignored if the row permission is already defined as enabled for row access control.

DISABLE

Specifies that the row permission is to be disabled for row access control. If row access control is not currently activated for the table, the row permission will remain ineffective when row access control is activated for the table. If row access control is currently activated for the table, the row permission

becomes ineffective immediately and all packages and dynamic cached statements that reference the table are invalidated.

DISABLE is ignored if the row permission is already defined as disabled for row access control.

REGENERATE
Specifies that the row permission is to be regenerated. The row permission definition in the catalog is used, and existing authorizations and dependencies, if any, are retained. The user-defined functions that are referenced in the row permission definition must be resolved to the same secure UDFs as that were resolved during the row permission creation.

Notes

Applying DB2 maintenance:
When DB2 maintenance is applied that affects how a row permission is generated, the row permission might need to be regenerated to ensure the row permission is still valid.

If the row permission is regenerated successfully, the status of the row permission is set to a blank in the catalog table. If the row permission is enabled and row access control is currently activated for the table, all packages and dynamic cached statements that reference the table are invalidated.

If the row permission cannot be regenerated successfully, an error is returned. The regeneration status of the row permission is set to an error. If the row permission is enabled and row access control is currently activated for the table, all packages and dynamic cached statements that reference the table are marked invalidated. To clear the status of the row permission, the row permission must be dropped and re-created with a modified definition. Or the row permission can be disabled if not disabled yet. A disabled row permission becomes ineffective to a row access control enforced table.

When the table is referenced in a data manipulation statement, the statement returns an error if any enabled row permission has an regeneration error.

Examples

Example 1:
Enable permission P1.

```
ALTER PERMISSION P1 ENABLE;
```

Example 2:
Regenerate permission P1.

```
ALTER PERMISSION P1 REGENERATE;
```

ALTER PROCEDURE (external)

The ALTER PROCEDURE statement changes the description of an external stored procedure at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set that is defined below must include at least one of the following:
- Ownership of the stored procedure
- The ALTERIN privilege on the schema
- SYSADM or SYSCTRL authority
- System DBADM

The authorization ID that matches the schema name implicitly has the ALTERIN privilege on the schema.

If the authorization ID that is used to alter the procedure has installation SYSADM authority, the procedure is identified as system-defined procedure when the procedure definition is reevaluated.

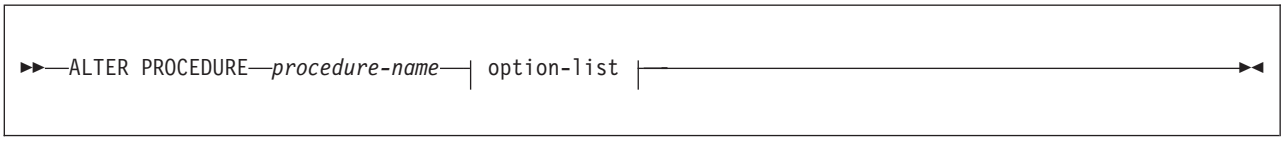
When **LANGUAGE** is **JAVA** and a *jar-name* is specified in the **EXTERNAL NAME** clause, the privilege set must include USAGE on the JAR file, the Java archive file.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the package.

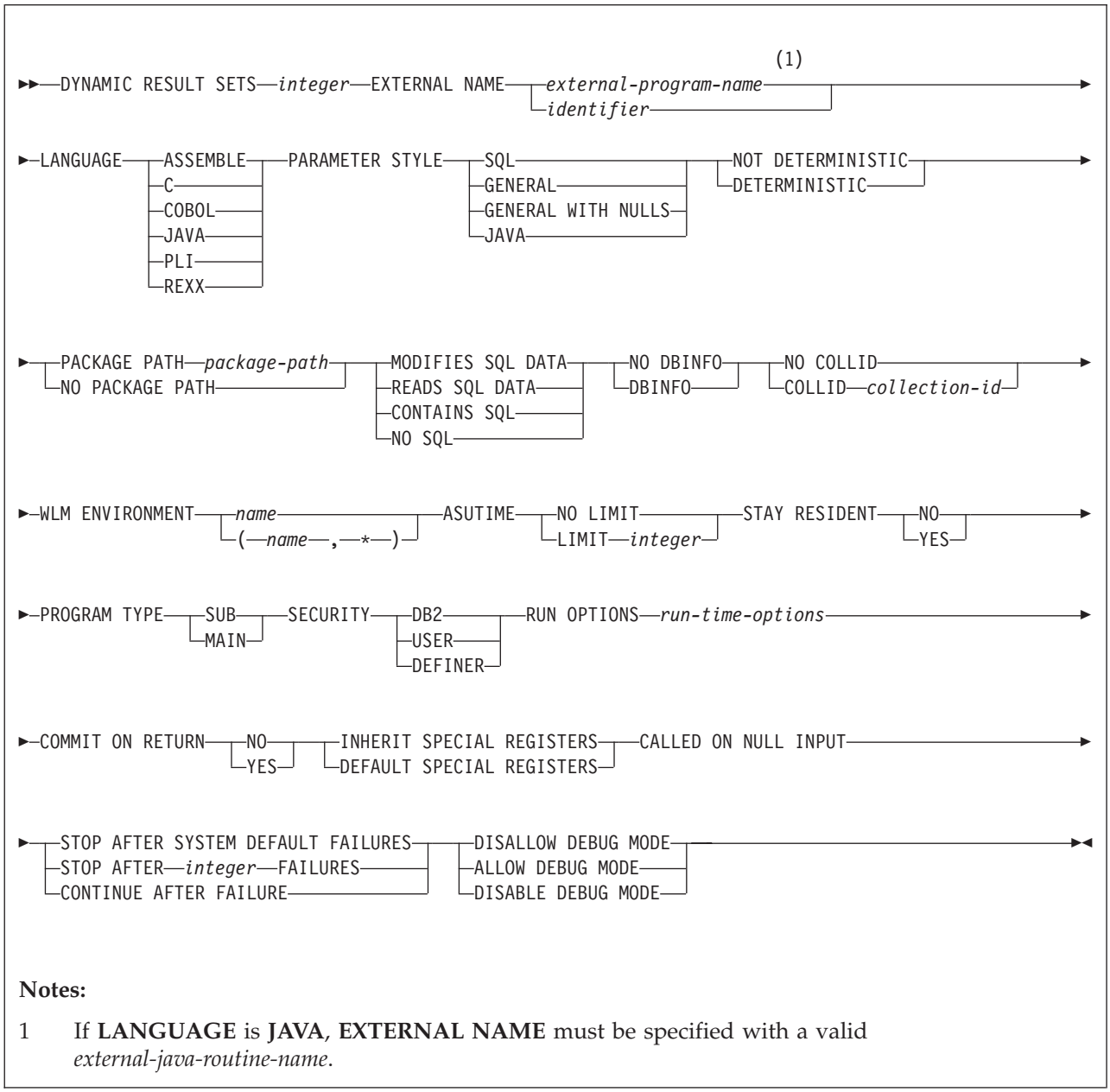
If the statement is dynamically prepared, the privilege set is the set of privileges that are held by the SQL authorization IDs of the process. The specified routine name can include a schema name (a qualifier). However, if the schema name is not the same as one of these SQL authorization IDs, one of the following conditions must be met:
- The privilege set includes SYSADM authority
- The privilege set includes SYSCTRL authority
- The SQL authorization ID of the process has the ALTERIN privilege on the schema

If the environment in which the stored procedure is to run is being changed, the authorization ID must have authority to use the WLM environment. This authorization is obtained from an external security product, such as RACF.

Syntax



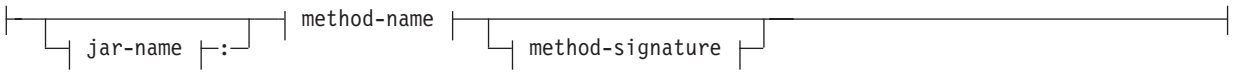
option-list: (Specify options in any order. Specify at least one option. Do not specify the same option more than once.)



Notes:

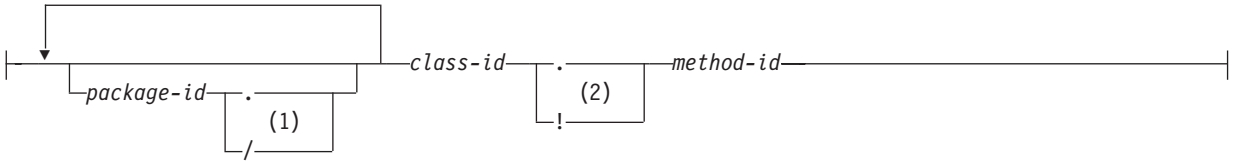
1 If **LANGUAGE** is **JAVA**, **EXTERNAL NAME** must be specified with a valid *external-java-routine-name*.

external-java-routine-name:



jar-name:

method-name:



method-signature:

Notes:

1 The slash (/) is supported for compatibility with previous releases of DB2 for z/OS.

2 The exclamation point (!) is supported for compatibility with other products in the DB2 family.

Description

procedure-name
Identifies the stored procedure to be altered.

DYNAMIC RESULT SETS *integer*
Specifies the maximum number of query result sets that the stored procedure can return. The value must be between 0 and 32767.

EXTERNAL NAME *external-program-name* **or** *identifier*
Specifies the name of the MVS load module for the program that runs when the procedure name is specified in an SQL CALL statement.

If **LANGUAGE** is **JAVA**, *external-program-name* must be specified and enclosed in single quotation marks, with no extraneous blanks within the single quotation marks. It must specify a valid *external-java-routine-name*. If multiple *external-program-name* values are specified, the total length of all of the values must not be greater than 1305 bytes and each value must be separated by a space or a line break. Do not specify a JAR file for a Java procedure for which **NO SQL** is in effect.

An *external-java-routine-name* contains the following parts:

jar-name
Identifies the name given to the JAR file when it was installed in the database. The name contains *jar-id*, which can optionally be qualified with a schema. Examples are "myJar" and "mySchema.myJar." The unqualified *jar-id* is implicitly qualified with a schema name according to the following rules:

- If the statement is embedded in a program, the schema name is the authorization ID in the **QUALIFIER** bind option when the package or plan was created or last rebound. If the **QUALIFIER** was not specified, the schema name is the owner of the package or plan.
- If the statement is dynamically prepared, the schema name is the SQL authorization ID in the CURRENT SCHEMA special register.

If *jar-name* is specified, it must exist when the ALTER PROCEDURE statement is processed.

If *jar-name* is not specified, the procedure is loaded from the class file directly instead of being loaded from a JAR file. DB2 searches the directories in the CLASSPATH associated with the WLM Environment. Environmental variables for Java routines are specified in a data set identified in a JAVAENV DD card on the JCL used to start the address space for a WLM-managed stored procedure.

method-name
Identifies the name of the method and must not be longer than 254 bytes. Its package, class, and method ID's are specific to Java and as such are not limited to 18 bytes. In addition, the rules for what these can contain are not necessarily the same as the rules for an SQL ordinary identifier.

package-id
Identifies a package. The concatenated list of *package-ids* identifies the package that the class identifier is part of. If the class is part of a package, the method name must include the complete package prefix, such as "myPacks.StoredProcs." The Java virtual machine looks in the directory "/myPacks/StoredProcs/" for the classes.

class-id
Identifies the class identifier of the Java object.

method-id
Identifies the method identifier with the Java class to be invoked.

method-signature
Identifies a list of zero or more Java data types for the parameter list and must not be longer than 1024 bytes. Specify the *method-signature* if the procedure involves any input or output parameters that can be NULL. When the stored procedure being created is called, DB2 searches for a Java method with the exact *method-signature*. The number of *java-datatype* elements specified indicates how many parameters that the Java method must have.

A Java procedure can have no parameters. In this case, you code an empty set of parentheses for *method-signature*. If a Java *method-signature* is not specified, DB2 searches for a Java method with a signature derived from the default JDBC types associated with the SQL types specified in the parameter list of the ALTER PROCEDURE statement.

For other values of **LANGUAGE**, the value must conform to the naming conventions for MVS load modules: the value must be less than or equal to 8

bytes, and it must conform to the rules for an ordinary identifier with the exception that it must not contain an underscore.

LANGUAGE
Specifies the application programming language in which the stored procedure is written. Assembler, C, COBOL, and PL/I programs must be designed to run in IBM's Language Environment.

ASSEMBLE
The stored procedure is written in Assembler.

C The stored procedure is written in C or C++.

COBOL
The stored procedure is written in COBOL, including the OO-COBOL language extensions.

JAVA
The stored procedure is written in Java and is executed in the Java Virtual Machine. When **LANGUAGE JAVA** is specified, the **EXTERNAL NAME** clause must also be specified with a valid *external-java-routine-name* and **PARAMETER STYLE** must be specified with **JAVA**. The procedure must be a public static method of the specified Java class.

Do not specify **LANGUAGE JAVA** when **DBINFO**, **PROGRAM TYPE MAIN**, or **RUN OPTIONS** is in effect.

PLI
The stored procedure is written in PL/I.

REXX
The stored procedure is written in REXX. Do not specify **LANGUAGE REXX** when **PARAMETER STYLE SQL** is specified.

PARAMETER STYLE
Identifies the linkage convention used to pass parameters to and return values from the stored procedure. All of the linkage conventions provide arguments to the stored procedure that contain the parameters specified on the CALL statement. Some of the linkage conventions pass additional arguments to the stored procedure that provide more information to the stored procedure. For more information on linkage conventions, see *DB2 Application Programming and SQL Guide*.

SQL
Specifies that, in addition to the parameters on the CALL statement, several additional parameters are passed to the stored procedure. The following parameters are passed:

- The first *n* parameters that are specified on the CREATE PROCEDURE statement.
- *n* parameters for indicator variables for the parameters.
- The SQLSTATE to be returned.
- The qualified name of the stored procedure.
- The specific name of the stored procedure.
- The SQL diagnostic string to be returned to DB2.
- If **DBINFO** is specified, the **DBINFO** structure.

Do not specify **PARAMETER STYLE SQL** when **LANGUAGE REXX** is specified.

GENERAL
Specifies that the stored procedure uses a parameter passing mechanism where the stored procedure receives only the parameters specified on the CALL statement. Arguments to procedures defined with this parameter style cannot be null.

GENERAL WITH NULLS
Specifies that, in addition to the parameters on the CALL statement as specified in **GENERAL**, another argument is also passed to the stored procedure. The additional argument contains an indicator array with an element for each of the parameters on the CALL statement. In C, this is an array of short integers. The indicator array enables the stored procedure to accept or return null parameter values.

JAVA
Specifies that the stored procedure uses a parameter passing convention that conforms to the Java and SQLJ Routines specifications. **PARAMETER STYLE JAVA** can be specified only if **LANGUAGE** is **JAVA**. If the ALTER PROCEDURE statement results in changing **LANGUAGE** to **JAVA**, **PARAMETER STYLE JAVA**, and an **EXTERNAL NAME** clause might need to be specified to provide appropriate values. **JAVA** must be specified for **PARAMETER STYLE** when **LANGUAGE** is **JAVA**.

INOUT and **OUT** parameters are passed as single-entry arrays. The **INOUT** and **OUT** parameters are declared in the Java method as single-element arrays of the Java type.

PARAMETER STYLE SQL cannot be used with **LANGUAGE REXX**.

DETERMINISTIC or NOT DETERMINISTIC
Specifies whether the stored procedure returns the same results each time the stored procedure is called with the same **IN** and **INOUT** arguments.

DETERMINISTIC
The stored procedure always returns the same results each time the stored procedure is called with the same **IN** and **INOUT** arguments, if the referenced data in the database has not changed.

NOT DETERMINISTIC
The stored procedure might not return the same result each time the procedure is called with the same **IN** and **INOUT** arguments, even when the referenced data in the database has not changed.

DB2 does not verify that the stored procedure code is consistent with the specification of **DETERMINISTIC** or **NOT DETERMINISTIC**.

NO PACKAGE PATH or PACKAGE PATH *package-path*
Identifies the package path to use when the procedure is run. This is the list of the possible package collections into which the DBRM this is associated with the procedure is bound.

NO PACKAGE PATH
Specifies that the list of package collections for the procedure is the same as the list of package collections for the calling program. If the calling program does not use a package, DB2 resolves the package by using the CURRENT PACKAGE PATH special register, the CURRENT PACKAGESET special register, or the **PKLIST** bind option (in this order). For information about how DB2 uses these three items, see *DB2 Application Programming and SQL Guide*.

PACKAGE PATH *package-path*
Specifies a list of package collections, in the same format as used in the CURRENT PACKAGE PATH special register.

If the **COLLID** clause is specified with **PACKAGE PATH**, the **COLLID** clause is ignored when the routine is invoked.

The *package-path* value that is associated with the procedure definition is checked when the procedure is invoked. If *package-path* contains SESSION_USER, USER, PATH, or PACKAGE PATH, an error is returned when the *package-path* value is checked.

MODIFIES SQL DATA, READS SQL DATA, CONTAINS SQL, or NO SQL
Specifies which SQL statements, if any, can be executed in the procedure or any routine that is called from this procedure. For the data access classification of each statement, see Table 162 on page 2030.

MODIFIES SQL DATA
Specifies that the procedure can execute any SQL statement except statements that are not supported in procedures.

READS SQL DATA
Specifies that procedure can execute statements with a data access indication of READS SQL DATA, CONTAINS SQL, or NO SQL. The procedure cannot execute SQL statements that modify data.

CONTAINS SQL
Specifies that the procedure can execute only SQL statements with an access indication of CONTAINS SQL. The procedure cannot execute statements that read or modify data.

NO SQL
Specifies that the procedure can execute only SQL statements with a data access classification of NO SQL. Do not specify **NO SQL** for a Java procedure that uses a JAR file.

NO DBINFO or DBINFO
Specifies whether additional status information is passed to the stored procedure when it is invoked.

NO DBINFO
Additional information is not passed.

DBINFO
An additional argument is passed when the stored procedure is invoked. The argument is a structure that contains information such as the application run time authorization ID, the schema name, the name of a table or column that the procedure might be inserting into or updating, and identification of the database server that invoked the procedure. For details about the argument and its structure, see *DB2 Application Programming and SQL Guide*.

DBINFO can be specified only if **PARAMETER STYLE SQL** is specified.

NO COLLID or COLLID *collection-id*
Identifies the package collection that is to be used when the stored procedure is executed. This is the package collection into which the DBRM that is associated with the stored procedure is bound.

NO COLLID
Specifies that the package collection for the stored procedure is the same as the package collection of the calling program. If the invoking program does

not use a package, DB2 resolves the package by using the CURRENT PACKAGE PATH special register, the CURRENT PACKAGESET special register, or the **PKLIST** bind option (in this order). For details about how DB2 uses these three items, see the information on package resolution in *DB2 Application Programming and SQL Guide*.

COLLID *collection-id*
Identifies the package collection that is to be used when the stored procedure is executed. It is the name of the package collection into which the DBRM associated with the stored procedure is bound.

For REXX stored procedures, *collection-id* can be DSNREXRR, DSNREXRS, DSNREXCR, or DSNREXCS.

WLM ENVIRONMENT
Identifies the WLM (workload manager) environment in which the stored procedure is to run when the DB2 stored procedure address space is WLM-established. The *name* of the WLM environment is an SQL identifier.

name
The WLM environment in which the stored procedure must run. If another stored procedure or a user-defined function calls the stored procedure and that calling routine is running in an address space that is not associated with the specified WLM environment, DB2 routes the stored procedure request to a different address space.

(*name***,*)**
When the stored procedure is called directly by an SQL application program, the WLM environment in which the stored procedure runs.

If another stored procedure or a user-defined function calls the stored procedure, the stored procedure runs in the same WLM environment that the calling routine uses.

To change the environment in which the procedure is to run, you must have appropriate authority for the WLM environment. For an example of a RACF command that provides this authorization, see Running stored procedures.

ASUTIME
Specifies the total amount of processor time, in CPU service units, that a single invocation of a stored procedure can run. The value is unrelated to the ASUTIME column in the resource limit specification table.

When you are debugging a stored procedure, setting a limit can be helpful in case the stored procedure gets caught in a loop. For information on CPU service units, see *z/OS MVS Initialization and Tuning Guide*.

NO LIMIT
There is no limit on the service units.

LIMIT *integer*
The limit on the service units is a positive *integer* in the range of 1 to 2 147 483 647. If the stored procedure uses more service units than the specified value, DB2 cancels the stored procedure.

STAY RESIDENT
Specifies whether the stored procedure load module is to remain resident in memory when the stored procedure ends.

NO The load module is deleted from memory after the stored procedure ends. Use **NO** for non-reentrant stored procedures.

YES
The load module remains resident in memory after the stored procedure ends.

PROGRAM TYPE
Specifies whether the stored procedure runs as a main routine or a subroutine. If PROGRAM TYPE is altered, the stored procedure needs to be re-compiled for the change to take effect.

SUB
The stored procedure runs as a subroutine.

Do not specify **PROGRAM TYPE SUB** for stored procedures with a **LANGUAGE** value of **REXX**.

MAIN
The stored procedure runs as a main routine.

Do not specify **PROGRAM TYPE MAIN** when **LANGUAGE JAVA** is specified.

SECURITY
Specifies how the stored procedure interacts with an external security product, such as RACF, to control access to non-SQL resources.

DB2
The stored procedure does not require a special external security environment. If the stored procedure accesses resources that an external security product protects, the access is performed using the authorization ID associated with the stored procedure address space.

USER
An external security environment should be established for the stored procedure. If the stored procedure accesses resources that the external security product protects, the access is performed using the authorization ID of the user who invoked the stored procedure.

DEFINER
An external security environment should be established for the stored procedure. If the stored procedure accesses resources that the external security product protects, the access is performed using the authorization ID of the owner of the stored procedure.

RUN OPTIONS *run-time-options*
Specifies the Language Environment run time options to be used for the stored procedure. For a REXX stored procedure, specifies the Language Environment run time options to be passed to the REXX language interface to DB2. You must specify *run-time-options* as a character string that is no longer than 254 bytes. To replace any existing run time options with no options, specify an empty string with **RUN OPTIONS**. When you specify an empty string, DB2 does not pass any run time options to Language Environment, and Language Environment uses its installation defaults. For a description of the Language Environment run time options, see *z/OS Language Environment Programming Reference*.

Do not specify **RUN OPTIONS** when **LANGUAGE JAVA** is specified.

COMMIT ON RETURN
Indicates whether DB2 is to commit the transaction immediately on return from the stored procedure.

NO DB2 does not issue a commit when the stored procedure returns.

YES
DB2 issues a commit when the stored procedure returns if the following statements are true:
- The SQLCODE that is returned by the CALL statement is not negative.
- The stored procedure is not in a must abort state.

The commit operation includes the work that is performed by the calling application process and the stored procedure.

If the stored procedure returns result sets, the cursors that are associated with the result sets must have been defined WITH HOLD to be usable after the commit.

INHERIT SPECIAL REGISTERS or DEFAULT SPECIAL REGISTERS
Specifies how special registers are set on entry to the routine.

INHERIT SPECIAL REGISTERS
Indicates that values of special registers are inherited according to the rules listed in the table for characteristics of special registers in a stored procedure in Table 40 on page 205.

DEFAULT SPECIAL REGISTERS
Indicates that special registers are initialized to the default values, as indicated by the rules in the table for characteristics of special registers in a stored procedure in Table 40 on page 205.

CALLED ON NULL INPUT
Specifies that the procedure is to be called even if any or all of the argument values are null, which means that the procedure must be coded to test for null argument values. The procedure can return null or nonnull values.

STOP AFTER SYSTEM DEFAULT FAILURES, STOP AFTER *nn* FAILURES, or CONTINUE AFTER FAILURE
Specifies whether the routine is to be put in a stopped state after some number of failures.

STOP AFTER SYSTEM DEFAULT FAILURES
Specifies that this routine should be placed in a stopped state after the number of failures indicated by the value of field MAX ABEND COUNT on installation field DSNTIPX.

STOP AFTER *nn* FAILURES
Specifies that this routine should be placed in a stopped state after *nn* failures. The value *nn* can be an integer from 1 to 32767.

CONTINUE AFTER FAILURE
Specifies that this routine should not be placed in a stopped state after any failure.

ALLOW DEBUG MODE, DISALLOW DEBUG MODE, or DISABLE DEBUG MODE
Specifies whether the procedure can be run in debugging mode.

Do not specify this option unless the procedure is defined with **LANGUAGE JAVA**.

ALLOW DEBUG MODE
Specifies that the procedure can be run in debugging mode.

DISALLOW DEBUG MODE
Specifies that the procedure cannot be run in debugging mode.

You can use a subsequent ALTER PROCEDURE statement to change this option to **ALLOW DEBUG MODE**.

DISABLE DEBUG MODE
Specifies that the procedure can never be run in debugging mode.

The procedure cannot be changed to specify **ALLOW DEBUG MODE** or **DISALLOW DEBUG MODE** when the procedure has been created or altered to use **DISABLE DEBUG MODE**. To change this option, you must drop and re-create the procedure using the option that you want.

Notes

Invalidation of packages: When an external procedure is altered, all the packages that refer to that procedure are marked invalid.

LANGUAGE C ***and the*** **PARAMETER VARCHAR** ***clause:*** The ALTER PROCEDURE statement does not allow you to alter the value of the **PARAMETER VARCHAR** or **PARAMETER CCSID** clauses that are associated with the procedure definition. However, you can alter the **LANGUAGE** clause for the procedure. If the **PARAMETER VARCHAR** clause is specified for the creation of a LANGUAGE C procedure, the catalog information for that option is not affected by subsequent ALTER PROCEDURE statements. The procedure might be changed to a language other than C, in which case the **PARAMETER VARCHAR** setting is ignored. If the procedure is later changed back to LANGUAGE C, the setting of the **PARAMETER VARCHAR** option that was specified for the CREATE PROCEDURE statement (which is still in the catalog) will be used.

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords:

- DYNAMIC RESULT SET, RESULT SET, and RESULT SETS as synonyms for **DYNAMIC RESULT SETS**
- STANDARD CALL as a synonym for **DB2SQL**
- SIMPLE CALL as a synonym for **GENERAL**
- SIMPLE CALL WITH NULLS as a synonym for **GENERAL WITH NULLS**
- VARIANT as a synonym for **NOT DETERMINISTIC**
- NOT VARIANT as a synonym for **DETERMINISTIC**
- NULL CALL as a synonym for **CALLED ON NULL INPUT**
- PARAMETER STYLE DB2SQL as a synonym for **PARAMETER STYLE SQL**

Example

Assume that stored procedure SYSPROC.MYPROC is currently defined to run in WLM environment PARTSA and that you have appropriate authority on that WLM environment and WLM environment PARTSEC. Change the definition of the stored procedure so that it runs in PARTSEC.

```
ALTER PROCEDURE SYSPROC.MYPROC WLM ENVIRONMENT PARTSEC;
```

ALTER PROCEDURE (SQL - external)

The ALTER PROCEDURE statement changes the description, at the current server, of an external SQL procedure.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set that is defined below must include at least one of the following:
- Ownership of the stored procedure
- The ALTERIN privilege on the schema
- SYSADM or SYSCTRL authority
- System DBADM

The authorization ID that matches the schema name implicitly has the ALTERIN privilege on the schema.

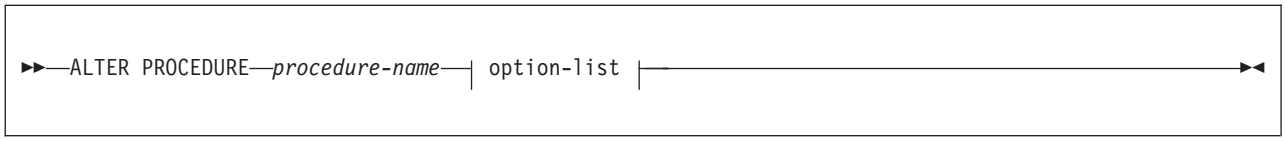
If the authorization ID that is used to alter the procedure has installation SYSADM authority, the procedure is identified as system-defined procedure when the procedure definition is reevaluated.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the package.

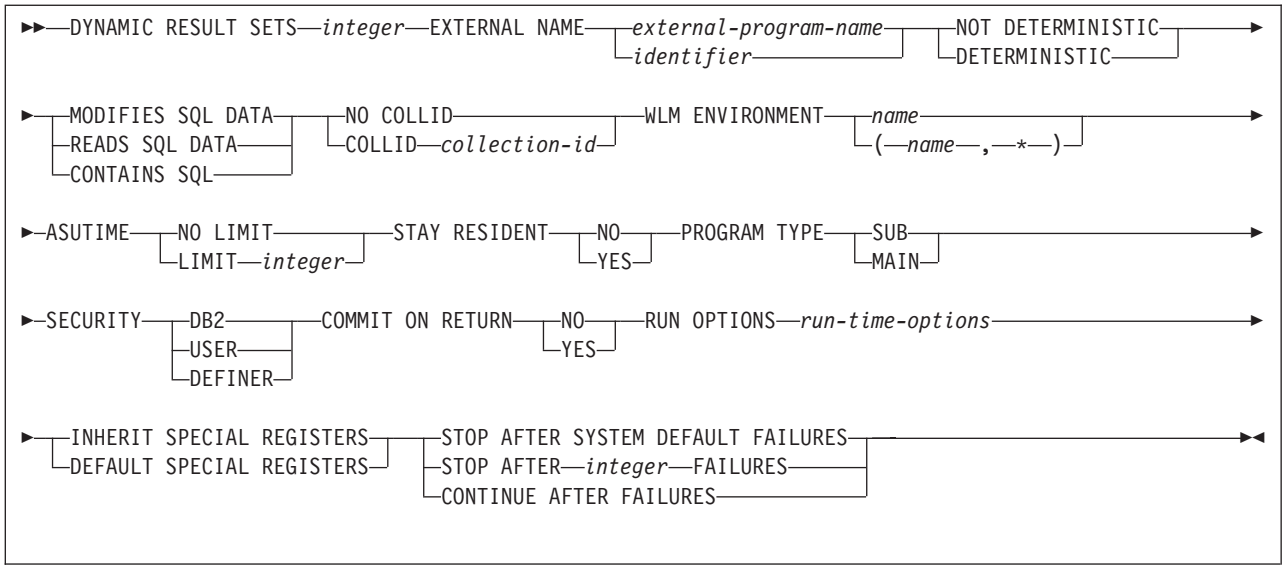
If the statement is dynamically prepared, the privilege set is the set of privileges that are held by the SQL authorization IDs of the process. The specified routine name can include a schema name (a qualifier). However, if the schema name is not the same as one of these SQL authorization IDs, one of the following conditions must be met:
- The privilege set includes SYSADM authority
- The privilege set includes SYSCTRL authority
- The SQL authorization ID of the process has the ALTERIN privilege on the schema

The SQL authorization ID that is used to alter the procedure definition must have appropriate authority for the WLM environment in which the procedure is currently defined to run. This authorization is obtained from an external security product, such as RACF.

Syntax



option-list: (Specify options in any order. Specify at least one option. Do not specify the same option more than once.)



Description

procedure-name
Identifies the stored procedure to be altered.

DYNAMIC RESULT SETS *integer*
Specifies the maximum number of query result sets that the procedure can return. The value must be between 0 and 32767.

EXTERNAL NAME *external-program-name* **or** *identifier*
Specifies the name of the MVS load module for the program that runs when the procedure name is specified in an SQL CALL statement. The value must conform to the naming conventions for MVS load modules: the value must be less than or equal to 8 bytes, and it must conform to the rules for an ordinary identifier with the exception that it must not contain an underscore.

NOT DETERMINISTIC or DETERMINISTIC
Specifies whether the procedure returns the same results each time the procedure is called with the same IN and INOUT arguments.

NOT DETERMINISTIC
The procedure might not return the same result each time the procedure is called with the same IN and INOUT arguments, even when the referenced data in the database has not changed.

DETERMINISTIC
The procedure always returns the same results each time the procedure is called with the same IN and INOUT arguments, if the referenced data in the database has not changed.

DB2 does not verify that the procedure code is consistent with the specification of **DETERMINISTIC** or **NOT DETERMINISTIC**.

MODIFIES SQL DATA, READS SQL DATA, or CONTAINS SQL
Specifies the classification of SQL statements that the procedure can execute. For the data access classification of each statement, see Table 162 on page 2030. Statements that are not supported in any procedure will return an error.

MODIFIES SQL DATA
Specifies that the procedure can execute any SQL statement except statements that are not supported in procedures.

READS SQL DATA
Specifies that procedure can execute statements with a data access indication of READS SQL DATA or CONTAINS SQL. The procedure cannot execute SQL statements that modify data.

CONTAINS SQL
Specifies that the procedure can execute only SQL statements with an access indication of CONTAINS SQL. The procedure cannot execute statements that read or modify data.

NO COLLID or COLLID *collection-id*
Identifies the package collection that is to be used when the procedure is executed. This is the package collection into which the DBRM that is associated with the procedure is bound.

NO COLLID
Indicates that the package collection for the procedure is the same as the package collection of the calling program. If the invoking program does not use a package, DB2 resolves the package by using the CURRENT PACKAGE PATH special register, the CURRENT PACKAGESET special register, or the **PKLIST** bind option (in this order). For details about how DB2 uses these three items, see the information on package resolution in *DB2 Application Programming and SQL Guide*.

COLLID *collection-id*
Specifies the package collection for the procedure.

WLM ENVIRONMENT *name* **or (***name***,*)**
Identifies the WLM (workload manager) environment in which the procedure is to run when the DB2 stored procedure address space is WLM-established. The *name* of the WLM environment is an SQL identifier.

name
Specifies the WLM environment in which the procedure must run. If another routine calls the procedure and that calling routine is running in an address space that is not associated with the specified WLM environment, DB2 routes the procedure request to a different address space.

(*name***,*)**
When an SQL application program directly calls a procedure, *name* specifies the WLM environment in which the stored procedure runs.

If another routine calls the procedure, the procedure runs in the same WLM environment that the calling routine uses.

To change the environment in which the procedure is to run, you must have appropriate authority for the WLM environment. For an example of a RACF command that provides this authorization, see Running stored procedures.

ASUTIME
Specifies the total amount of processor time, in CPU service units, that a single invocation of a procedure can run. The value is unrelated to the ASUTIME column of the resource limit specification table.

When you are debugging a procedure, setting a limit can be helpful in case the procedure gets caught in a loop. For information on service units, see *z/OS MVS Initialization and Tuning Guide*.

NO LIMIT
There is no limit on the number of CPU service units that the procedure can run.

LIMIT *integer*
The limit on the number of CPU service units is a positive *integer* in the range of 1 to 2 147 483 647. If the procedure uses more service units than the specified value, DB2 cancels the procedure. The CPU cycles that are consumed by parallel tasks in a procedure do not contribute towards the specified ASUTIME LIMIT.

STAY RESIDENT
Specifies whether the load module for the procedure is to remain resident in memory when the procedure ends.

NO The load module is deleted from memory after the procedure ends.

YES
The load module remains resident in memory after the procedure ends.

PROGRAM TYPE
Specifies whether the procedure runs as a main routine or a subroutine. If PROGRAM TYPE is altered, the stored procedure needs to be re-compiled for the change to take effect.

SUB
The procedure runs as a subroutine.

MAIN
The procedure runs as a main routine.

SECURITY
Specifies how the procedure interacts with an external security product, such as RACF, to control access to non-SQL resources.

DB2
The procedure does not require a special external security environment. If the procedure accesses resources that an external security product protects, the access is performed using the authorization ID that is associated with the address space in which the procedure runs.

USER
An external security environment should be established for the procedure. If the procedure accesses resources that the external security product protects, the access is performed using the authorization ID of the user who invoked the procedure.

DEFINER
An external security environment should be established for the procedure. If the procedure accesses resources that the external security product protects, the access is performed using the authorization ID of the owner of the procedure.

RUN OPTIONS *run-time-options*
Specifies the Language Environment run time options that are to be used for the procedure. You must specify *run-time-options* as a character string that is no longer than 254 bytes. If you do not specify **RUN OPTIONS** or pass an empty

string, DB2 does not pass any run time options to Language Environment, and Language Environment uses its installation defaults.

For a description of the Language Environment run time options, see *z/OS Language Environment Programming Reference*.

COMMIT ON RETURN
Indicates whether DB2 commits the transaction immediately on return from the procedure.

NO DB2 does not issue a commit when the procedure returns.

YES
DB2 issues a commit when the procedure returns if the following statements are true:
- A positive SQLCODE is returned by the CALL statement.
- The procedure is not in a must abort state.

The commit operation includes the work that is performed by the calling application process and the procedure.

If the procedure returns result sets, the cursors that are associated with the result sets must have been defined as WITH HOLD to be usable after the commit.

INHERIT SPECIAL REGISTERS or DEFAULT SPECIAL REGISTERS
Specifies how special registers are set on entry to the routine.

INHERIT SPECIAL REGISTERS
Specifies that special registers should be inherited according to the rules listed in the table for characteristics of special registers in a procedure in Table 40 on page 205.

DEFAULT SPECIAL REGISTERS
Specifies that special registers should be initialized to the default values, as indicated by the rules in the table for characteristics of special registers in a procedure in Table 40 on page 205.

STOP AFTER SYSTEM DEFAULT FAILURES, STOP AFTER *nn* FAILURES, or CONTINUE AFTER FAILURE
Specifies if the routine is stopped after failures.

STOP AFTER SYSTEM DEFAULT FAILURES
Specifies that this routine should be placed in a stopped state after the number of failures indicated by the value of field MAX ABEND COUNT on installation panel DSNTIPX.

STOP AFTER *nn* FAILURES
Specifies that this routine should be placed in a stopped state after *nn* failures. The value *nn* can be an integer from 1 to 32767.

CONTINUE AFTER FAILURES
Specifies that this routine should not be placed in a stopped state after any failure.

Notes

Changing to a native SQL procedure: You cannot change an external SQL procedure to a native SQL procedure. You can drop the procedure that you want to change using the DROP statement and create a native SQL procedure with a similar definition using the CREATE PROCEDURE statement. Alternatively, you can create a native SQL procedure using a different schema.

Invalidation of packages: When an SQL procedure is altered, all packages that refer to that procedure are marked invalid.

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 UDB family, DB2 supports the following keywords:

- RESULT SET, RESULT SETS, and DYNAMIC RESULT SET as synonyms for **DYNAMIC RESULT SETS**.
- VARIANT as a synonym for **NOT DETERMINISTIC**
- NOT VARIANT as a synonym for **DETERMINISTIC**

Example

Modify the definition for an SQL procedure so that SQL changes are committed on return from the SQL procedure and the SQL procedure runs in the WLM environment named WLMSQLP.

```
ALTER PROCEDURE UPDATE_SALARY_1
 COMMIT ON RETURN YES
 WLM ENVIRONMENT WLMSQLP;
```

ALTER PROCEDURE (SQL - native)

The ALTER PROCEDURE statement changes the definition of an SQL procedure at the current server. The procedure options, parameter names, and routine body can be changed and additional versions of the procedure can be defined and maintained using the ALTER PROCEDURE statement.

For information about the SQL control statements that are supported in native SQL procedures, refer to Chapter 6, "SQL control statements for SQL routines," on page 1963.

Invocation

This statement can only be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set that is defined below must include at least one of the following:
- Ownership of the procedure
- The ALTERIN privilege on the schema
- SYSADM authority
- SYSCTRL authority
- System DBADM

The authorization ID that matches the schema name implicitly has the ALTERIN privilege on the schema.

If the authorization ID that is used to alter the procedure has installation SYSADM authority, the procedure is identified as system-defined procedure when the procedure definition is reevaluated.

Additional privileges might be required in the following situations:
- If *SQL-routine-body* is specified, the privilege set must include the privileges that are required to execute the statements in *SQL-routine-body*.
- If a user-defined type is referenced (as the data type of a parameter or SQL variable), the privilege set must also include at least one of the following privileges or authorities:
  - Ownership of the user-defined type
  - The USAGE privilege on the user-defined type
  - SYSADM authority
- If the procedure uses a table as a parameter, the privilege set must also include at least one of the following privileges or authorities:
  - Ownership of the table
  - The SELECT privilege on the table
  - SYSADM authority
- If the WLM ENVIRONMENT FOR DEBUG MODE clause is specified, the privilege set must include the authority to define programs that run in the specified WLM environment. This authorization is obtained from an external security product, such as RACF.
- When defining a new version of a procedure (using the ADD VERSION clause) or when replacing an existing version (using the REPLACE VERSION clause), the privilege set must include the required authorization to add a new package

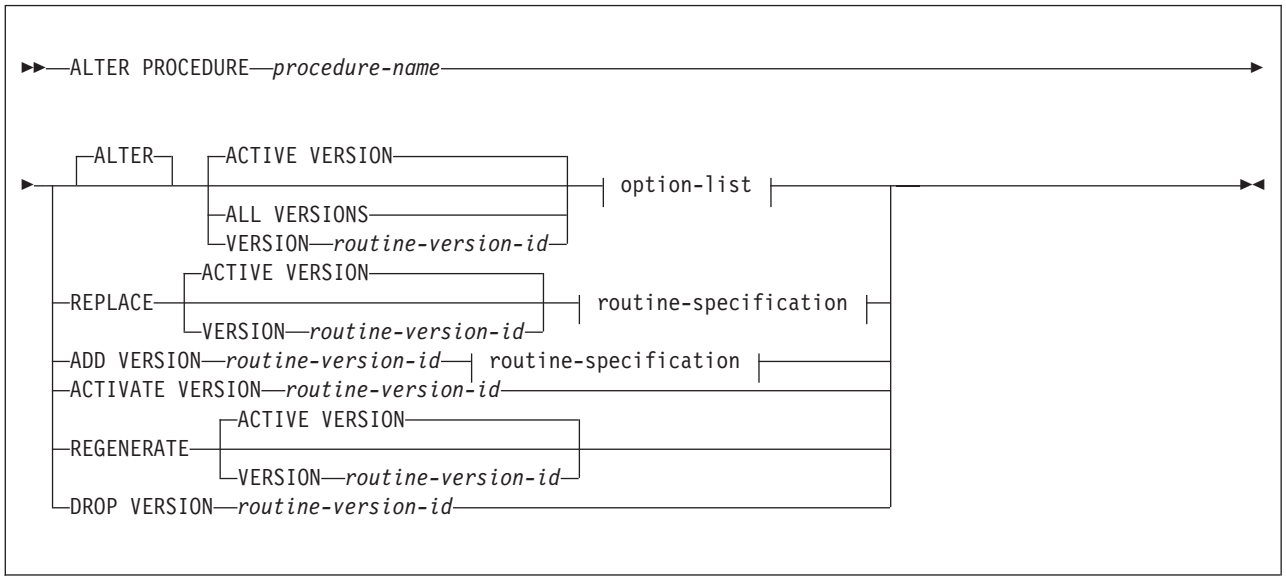
or a new version of an existing package depending on the value of the BIND NEW PACKAGE field on installation panel DSNTIPP, or the privilege set must include SYSADM or SYSCTRL authority.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the package.

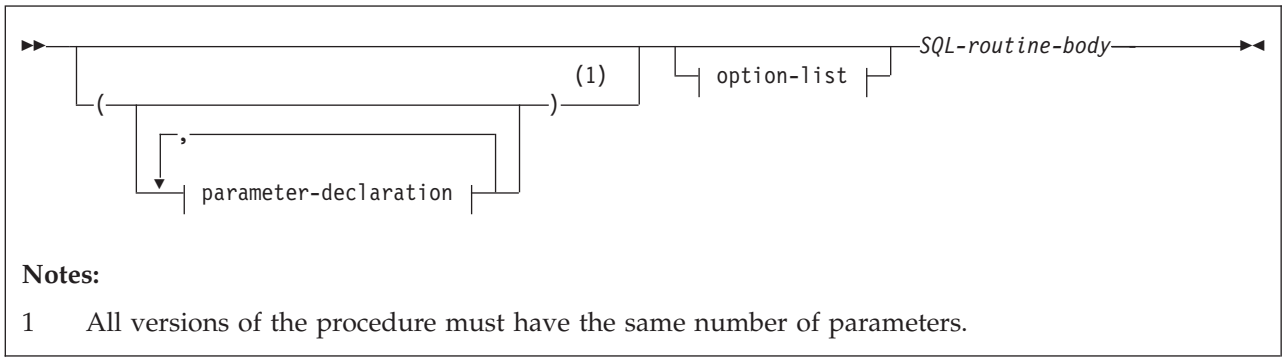
If the statement is dynamically prepared, the privilege set is the set of privileges that are held by the SQL authorization IDs of the process. The specified routine name can include a schema name (a qualifier). However, if the schema name is not the same as one of these SQL authorization IDs, one of the following conditions must be met:

- The privilege set includes SYSADM authority
- The privilege set includes SYSCTRL authority
- The SQL authorization ID of the process has the ALTERIN privilege on the schema

Syntax



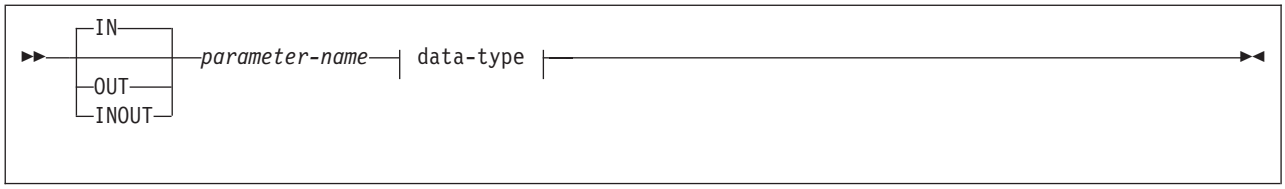
routine-specification:



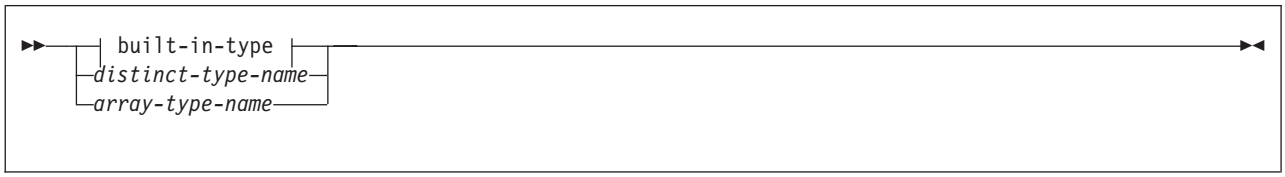
Notes:

1 All versions of the procedure must have the same number of parameters.

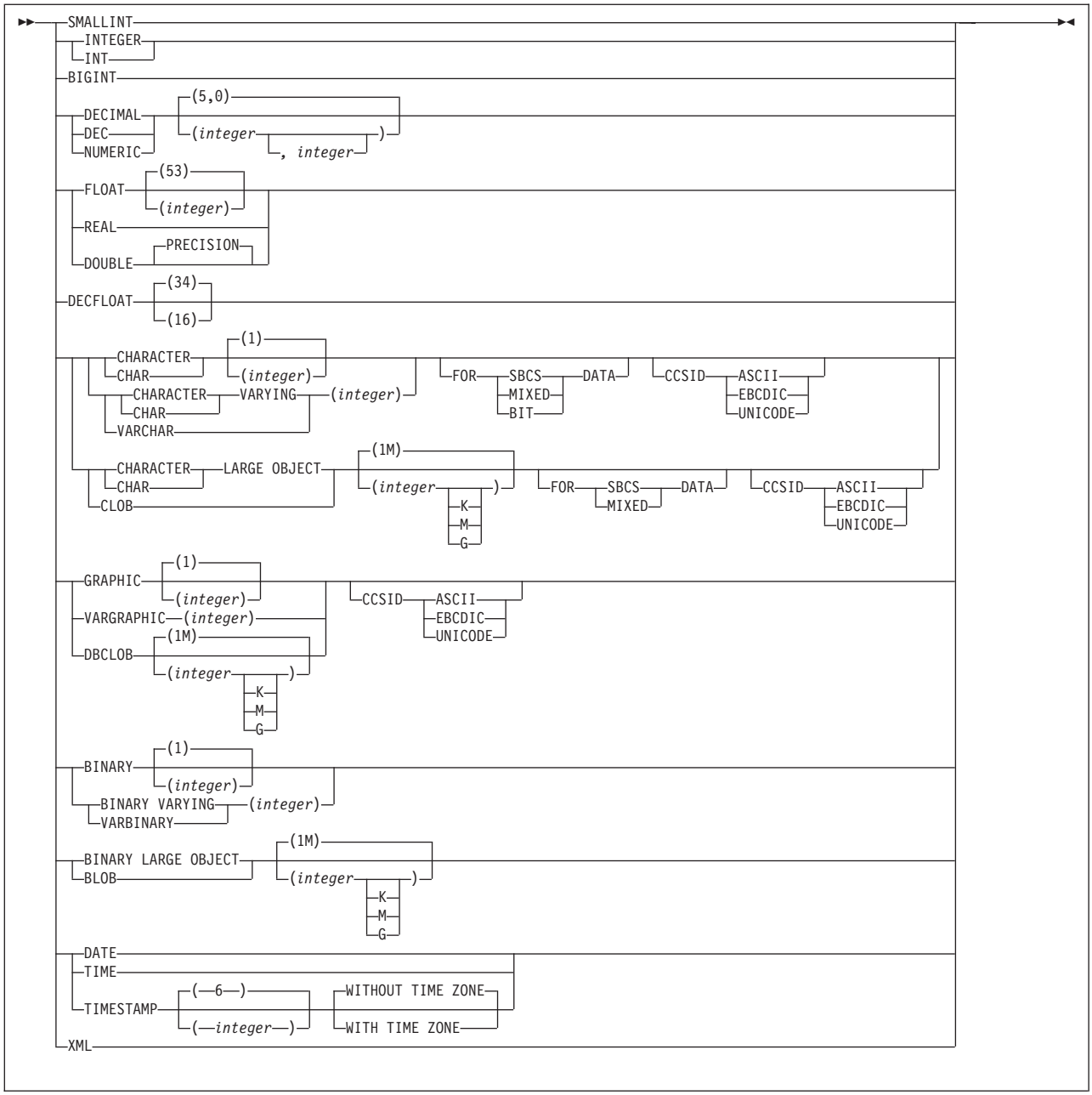
parameter-declaration:



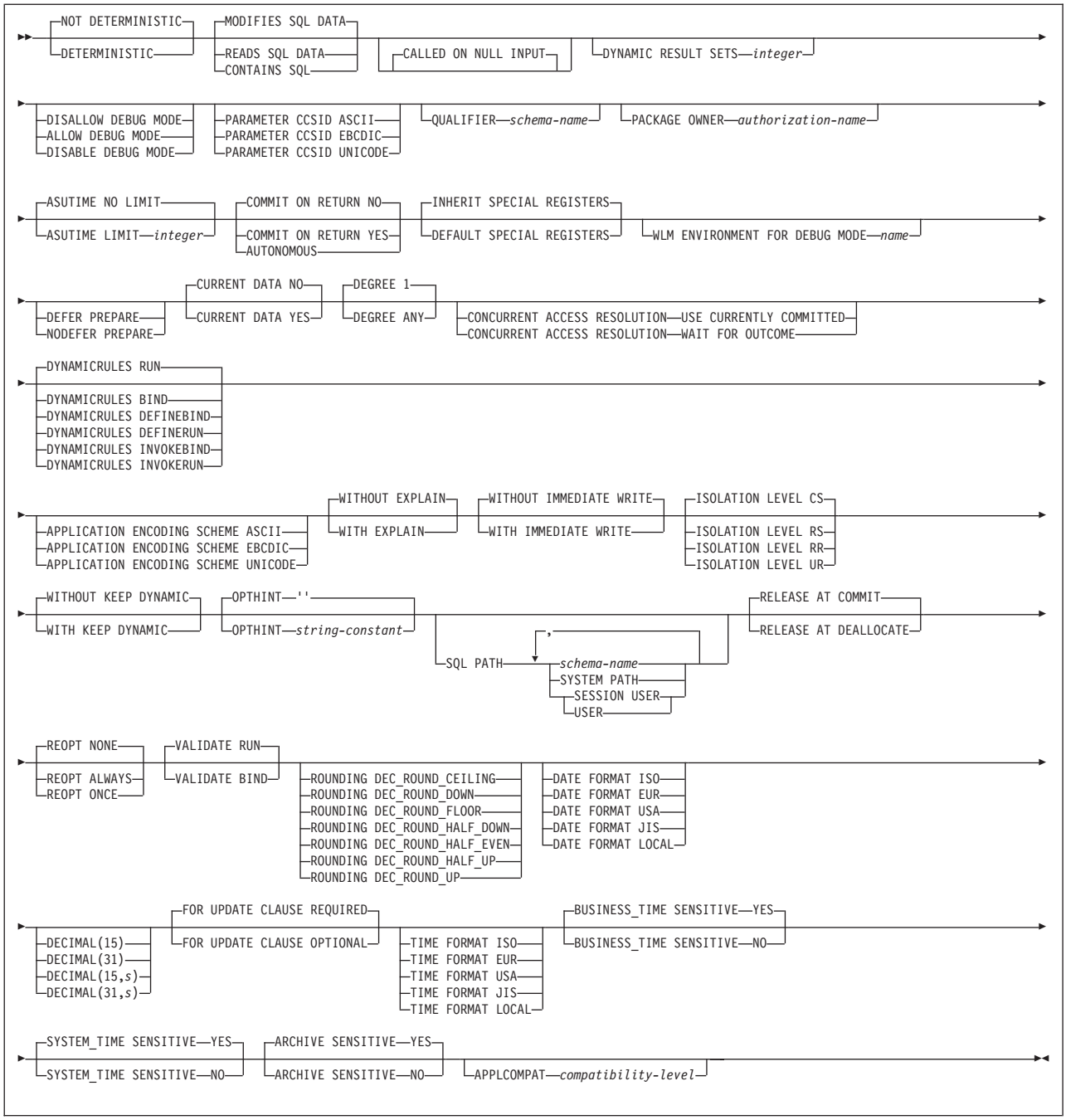
data-type:



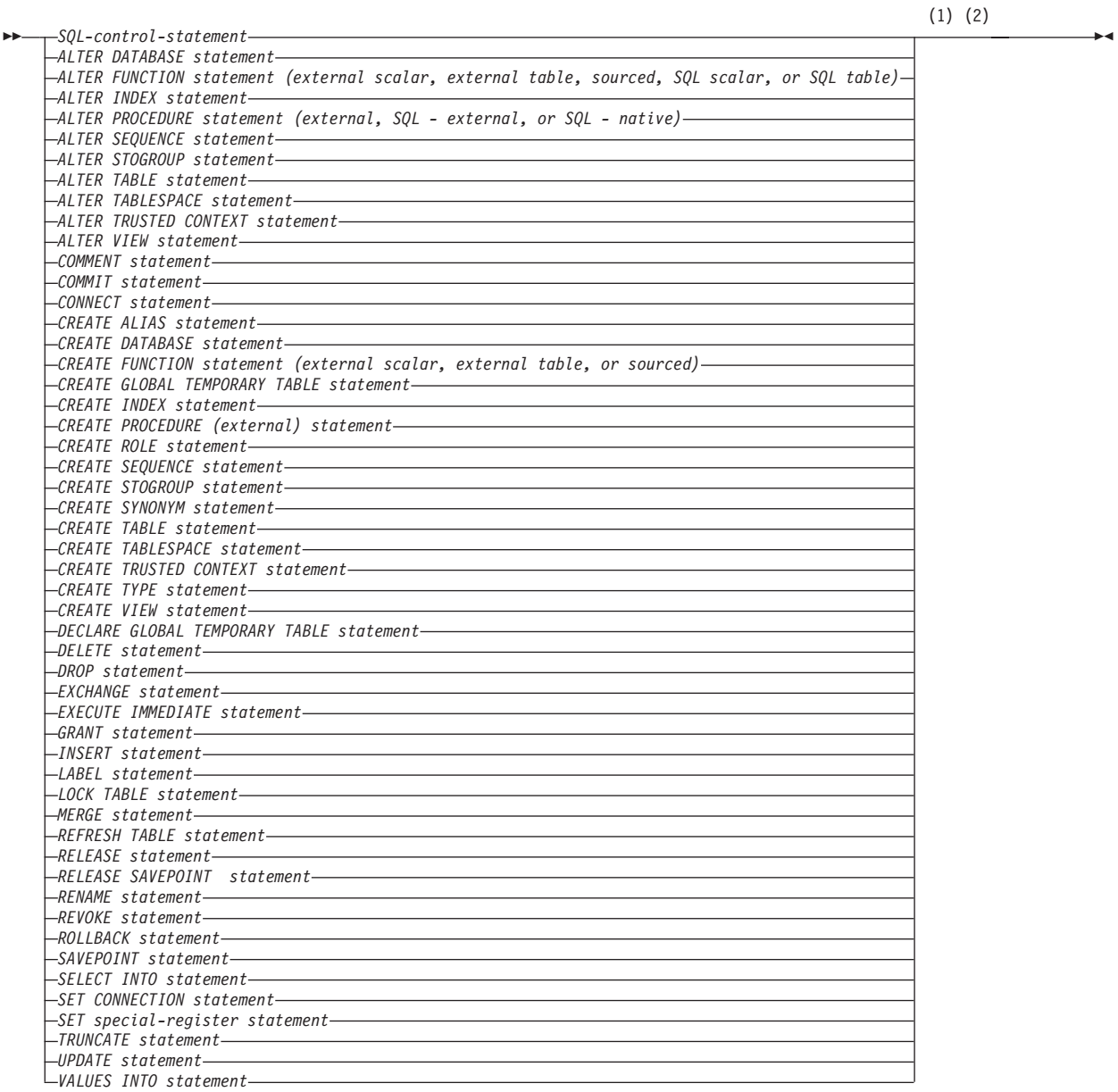
built-in-type:



option-list: (The options can be specified in any order, but each option can be specified only one time.)



SQL-routine-body:



Notes:

1 An ALTER FUNCTION (SQL scalar) statement or an ALTER PROCEDURE (SQL native) statement with an ADD VERSION or REPLACE clause is not allowed in an *SQL-routine-body*.

2 The COMMIT statement and the ROLLBACK statement (without the TO SAVEPOINT clause) must not be issued in a routine body if the routine is in the calling chain of an SQL routine, an external routine, or a trigger.

Description

procedure-name
Identifies the procedure to alter. The procedure that is identified in *procedure-name* must exist at the current server.

ACTIVE VERSION or ALL VERSIONS or VERSION *routine-version-id*
Identifies the version of the procedure that is to be changed, replaced, or regenerated depending on whether the ALTER, REPLACE, or REGENERATE keyword is specified.

ACTIVE VERSION
Specifies that the currently active version of the procedure is to be changed, replaced, or regenerated.

ALL VERSIONS
Specifies that all of the versions of the procedure are to be changed. Only the following options can be changed when this option is specified:

- AUTONOMOUS or COMMIT ON RETURN

VERSION *routine-version-id*
Identifies the version of the procedure that is to be changed, replaced, or regenerated. *routine-version-id* is the version identifier that is assigned when the version is defined. *routine-version-id* must identify a version of the specified procedure that exists at the current server.

ALTER
Specifies that a version of the procedure is to be changed.

When you change a procedure to add or replace a version of the procedure, any option that is not explicitly specified will use the existing value from the version of the procedure that is being changed.

REPLACE
Specifies that a version of the procedure is to be replaced.

Binding the replaced version of the procedure might result in a new access path even if the routine body is not changed.

When you replace a procedure, the data types, CCSID specifications, and character data attributes (FOR BIT/SBCS/MIXED DATA) of the parameters must be the same as the attributes of the corresponding parameters for the currently active version of the procedure. For options that are not explicitly specified, the system default values for those options are used, even if those options were explicitly specified for the version of the procedure that is being replaced. This is not the case for versions of the procedure that specified DISABLE DEBUG MODE. If DISABLE DEBUG MODE is specified for a version of a procedure, it cannot be changed by the REPLACE clause.

ADD VERSION *routine-version-id*
Specifies that a new version of the procedure is to be created. *routine-version-id* is the version identifier for the new version of the procedure. *routine-version-id* must not identify a version of the specified procedure that already exists at the current server.

When a new version of a procedure is created, the comment that is recorded in the catalog for the new version will be the same as the comment that is in the catalog for the currently active version.

When you add a new version of a procedure the data types, CCSID specifications, and character data attributes (FOR BIT/SBCS/MIXED DATA) of the parameters must be the same as the attributes of the corresponding parameters for the currently active version of the procedure. The parameter names can differ from the other versions of the procedure. For options that are not explicitly specified, the system default values will be used.

ACTIVATE VERSION *routine-version-id*
Specifies the version of the procedure that is to be the currently active version

of the procedure. *routine-version-id* is the version identifier that is assigned when the version of the procedure is defined. The version that is specified with *routine-version-id* is the version that will be invoked by the CALL statement, unless the value of the CURRENT ROUTINE VERSION special register overrides the currently active version of the procedure when the procedure is invoked. *routine-version-id* must identify a version of the procedure that already exists at the current server.

REGENERATE
Regenerates a version of the procedure. When DB2 maintenance is applied that changes how an SQL procedure is generated, the procedure might need to be regenerated to process the maintenance changes.

REGENERATE automatically rebinds, at the local server, the package for the SQL control statements for the procedure and rebinds the package for the SQL statements that are included in the procedure body. If a remote bind is also needed, the BIND PACKAGE COPY command must be explicitly done for all of the remote servers.

REGENERATE is different from a REBIND PACKAGE command where the SQL statements are rebound (i.e. to generate better access paths for those statements), but the SQL control statements in the procedure definition remain the same.

DROP VERSION *routine-version-id*
Drops the version of the procedure that is identified with *routine-version-id*. *routine-version-id* is the version identifier that is assigned when the version is defined. *routine-version-id* must identify a version of the procedure that already exists at the current server and must not identify the currently active version of the procedure. Only the identified version of the procedure is dropped.

When only a single version of the procedure exists at the current server, use the DROP PROCEDURE statement to drop the procedure. A version of the procedure for which the version identifier is the same as the contents of the CURRENT ROUTINE VERSION special register can be dropped if that version is not the currently active version of the procedure.

(*parameter-declaration***,...)**
Specifies the number of parameters of the procedure, the data type and usage of each parameter, and the name of each parameter for the version of the procedure that is being defined or changed. The number of parameters and the specified data type and usage of each parameter must match the data types in the corresponding position of the parameter for all other versions of this procedure. Synonyms for data types are considered to be a match.

IN, OUT, and INOUT specify the usage of the parameter. The usage of the parameters must match the implicit or explicit usage of the parameters of other versions of the same procedure.

IN Identifies the parameter as an input parameter to the procedure. The value of the parameter on entry to the procedure is the value that is returned to the calling SQL application, even if changes are made to the parameter within the procedure.

IN is the default.

OUT
Identifies the parameter as an output parameter that is returned by the procedure. If the parameter is not set within the procedure, the null value is returned.

INOUT
Identifies the parameter as both an input and output parameter for the procedure. If the parameter is not set within the procedure, its input value is returned.

parameter-name
Names the parameter for use as an SQL variable. The name cannot be the same as the name of any other *parameter-name* for this version of the procedure. The name of the parameter in this version of the procedure can be different than the name of the corresponding parameter for other versions of this procedure.

built-in-type
Specifies the data type of the parameter. See "CREATE PROCEDURE (SQL - native)" on page 1350 for more information on data type specifications.

distinct-type-name
The data type of the input parameter is a distinct type. Any length, precision, scale, subtype, or encoding scheme attributes for the parameter are those of the source type of the distinct type. The distinct type must not be based on a LOB data type.

array-type-name
The data type of the input parameter is a user-defined array type.

If you specify *array-type-name* without a schema name, DB2 resolves the array type by searching the schemas in the SQL path.

NOT DETERMINISTIC or DETERMINISTIC
Specifies whether the procedure returns the same results each time it is called with the same IN and INOUT arguments.

NOT DETERMINISTIC
The procedure might not return the same result each time it is called with the same IN and INOUT arguments, even when the data that is referenced in the database has not changed.

NOT DETERMINISTIC is the default.

DETERMINISTIC
The procedure always returns the same results each time it is called with the same IN and INOUT arguments if the data that is referenced in the database has not changed.

DB2 does not verify that the procedure code is consistent with the specification of DETERMINISTIC or NOT DETERMINISTIC.

MODIFIES SQL DATA, READS SQL DATA, or CONTAINS SQL
Specifies the classification of SQL statements that the procedure can execute.

MODIFIES SQL DATA
Specifies that the procedure can execute any SQL statement except statements that are not supported in procedures.

MODIFIES SQL DATA is the default.

READS SQL DATA
Specifies that procedure can execute statements with a data access indication of READS SQL DATA or CONTAINS SQL. The procedure cannot execute SQL statements that modify data.

CONTAINS SQL
Specifies that the procedure can execute only SQL statements with an access indication of CONTAINS SQL. The procedure cannot execute statements that read or modify data.

CALLED ON NULL INPUT
Specifies that the procedure will be called if any, or even if all parameter values are null.

DYNAMIC RESULT SETS *integer*
Specifies the maximum number of query result sets that the procedure can return. The default is DYNAMIC RESULT SETS 0, which indicates that there are no result sets. The value must be between 0 and 32767.

ALLOW DEBUG MODE, DISALLOW DEBUG MODE, or DISABLE DEBUG MODE
Specifies whether the version of the procedure can be run in debugging mode. The default for a new version of a procedure is determined using the value of the CURRENT DEBUG MODE special register.

ALLOW DEBUG MODE
Specifies that this version of the procedure can be run in debugging mode. When this version of the procedure is invoked and debugging is attempted, a WLM environment must be available.

DISALLOW DEBUG MODE
Specifies that the version of the procedure cannot be run in debugging mode.

You can use a subsequent ALTER PROCEDURE statement to change this option to ALLOW DEBUG MODE.

DISABLE DEBUG MODE
Specifies that the version of the procedure can never be run in debugging mode.

The version of the procedure cannot be changed to specify ALLOW DEBUG MODE or DISALLOW DEBUG MODE after the version of the procedure has been created, replaced, or altered to use DISABLE DEBUG MODE. To change DEBUG MODE for a version of a procedure that specifies DISABLE DEBUG MODE, you must drop and re-create the version of the procedure using the option that you want.

When DISABLE DEBUG MODE is in effect, the WLM ENVIRONMENT FOR DEBUG MODE option is ignored.

PARAMETER CCSID
Indicates whether the encoding scheme for character or graphic string parameters is ASCII, EBCDIC, or UNICODE. The default encoding scheme is the value that is specified in the CCSID clauses of the parameter list or in the field DEF ENCODING SCHEME on installation panel DSNTIPF.

This clause provides a convenient way to specify the encoding scheme for character or graphic string parameters. If individual CCSID clauses are specified for individual parameters in addition to this PARAMETER CCSID clause, the value that is specified in *all* of the CCSID clauses must be the same value that is specified in this clause.

If the data type for a parameter is a user-defined distinct type that is defined as a character or graphic type string, the CCSID of the distinct type must be the same as the value that is specified in this clause.

If the data type for a parameter is a user-defined array type that is defined with character or graphic string array elements, or a character string array index, the CCSID of these array attributes must be the same as the value that is specified in this clause.

This clause also specifies the encoding scheme that will be used for system-generated parameters of the routine.

QUALIFIER *schema-name*
Specifies the implicit qualifier that is used for unqualified names of tables, views, indexes, and aliases that are referenced in the procedure body. The default value is determined from the CURRENT SCHEMA special register.

PACKAGE OWNER *authorization-name*
Specifies the owner of the package that is associated with the version of the procedure. The SQL authorization ID of the process is the default value.

This authorization ID must have the privileges required to execute the SQL statements that are contained in the body of the routine and must contain the necessary bind privileges. The value of the PACKAGE OWNER option is subject to translation when sent to a remote system.

If the privilege set lacks SYSADM or SYSCTRL authority, *authorization-name* must be the same as one of the authorization IDs of the process. If the privilege set includes SYSADM or SYSCTRL authority, *authorization-name* can be any authorization ID that contains the necessary bind privileges.

ASUTIME
Specifies the total amount of processor time, in CPU service units, that a single invocation of a procedure can run. The value is unrelated to the ASUTIME column of the resource limit specification table.

When you are debugging a procedure, setting a limit can be helpful in case the procedure gets caught in a loop. For information on service units, see *z/OS MVS Initialization and Tuning Guide*.

NO LIMIT
Specifies that there is no limit on the number of CPU service units that the procedure can run.

NO LIMIT is the default.

LIMIT *integer*
The limit on the number of CPU service units is a positive *integer* in the range of 1 to 2 147 483 647. If the procedure uses more service units than the specified value, DB2 cancels the procedure. The CPU cycles that are consumed by parallel tasks in a procedure do not contribute towards the specified ASUTIME LIMIT.

COMMIT ON RETURN NO, COMMIT ON RETURN YES, or AUTONOMOUS
Indicates whether DB2 commits the transaction immediately on return from the procedure.

COMMIT ON RETURN NO
DB2 does not issue a commit when the procedure returns. NO is the default.

COMMIT ON RETURN YES,
DB2 issues a commit when the procedure returns if the following statements are true:

- The SQLCODE that is returned by the CALL statement is not negative.
- The procedure is not in a must-abort state.

The commit operation includes the work that is performed by the calling application process and by the procedure.

If the procedure returns result sets, the cursors that are associated with the result sets must have been defined as WITH HOLD to be usable after the commit.

AUTONOMOUS
DB2 executes the SQL procedure in a unit of work that is independent from the calling application. When this option is specified the procedure follows the rules of the COMMIT ON RETURN YES option before returning to the calling application. However, it does not commit changes in the calling application. When autonomous is specified:

- DYNAMIC RESULT SETS 0 must be in effect.
- Stored procedure parameters must not be defined as:
  - A LOB type
  - The XML data type
  - A distinct data type that is based on a LOB or XML value
  - An array type that is defined with array elements that are a LOB type

A value must not be assigned to a global variable when an autonomous procedure is executing.

INHERIT SPECIAL REGISTERS or DEFAULT SPECIAL REGISTERS
Specifies how special registers are set on entry to the routine.

INHERIT SPECIAL REGISTERS
Specifies that the values of special registers are inherited, according to the rules that are listed in the table for characteristics of special registers in a procedure in Table 40 on page 205.

INHERIT SPECIAL REGISTERS is the default.

DEFAULT SPECIAL REGISTERS
Specifies that special registers are initialized to the default values, as indicated by the rules in the table for characteristics of special registers in a procedure in Table 40 on page 205.

WLM ENVIRONMENT FOR DEBUG MODE *name*
Specifies the WLM (workload manager) application environment used by DB2 when debugging the procedure. The *name* of the WLM environment is an SQL identifier.

If you do not specify WLM ENVIRONMENT FOR DEBUG MODE, DB2 uses the default WLM-established stored procedure address space that is specified at installation time.

The WLM ENVIRONMENT FOR DEBUG MODE value is ignored when DISABLE DEBUG MODE is in effect.

To change the environment that DB2 uses for debugging this procedure, you must have the appropriate authority for the WLM application environment. For an example of a RACF command that provides this authorization, see Running stored procedures.

DEFER PREPARE or NODEFER PREPARE
Specifies whether to defer preparation of dynamic SQL statements that refer to remote objects, or to prepare them immediately.

The default depends on the value that is specified for the REOPT option. If REOPT NONE is specified, the default is NODEFER PREPARE. Otherwise, the default is DEFER PREPARE.

DEFER PREPARE
Specifies that the preparation of dynamic SQL statements that refer to remote objects will be deferred.

Refer to the DEFER(PREPARE) option in *DB2 Command Reference* for considerations with distributed processing.

NODEFER PREPARE
Specifies that the preparation of dynamic SQL statements that refer to remote objects will not be deferred.

CURRENT DATA
Specifies whether to require data currency for read-only and ambiguous cursors when the isolation level of cursor stability is in effect. CURRENT DATA also determines whether block fetch can be used for distributed, ambiguous cursors. For more information about updating the current row of a cursor, block fetch, and data currency, see *DB2 Application Programming and SQL Guide*.

YES
Specifies that data currency is required for read-only and ambiguous cursors. DB2 acquired page or row locks to ensure data currency. Block fetch is not allowed for distributed, ambiguous cursors.

NO Specifies that data currency is not required for read-only and ambiguous cursors. Block fetch is allowed for distributed, ambiguous cursors. Use of CURRENT DATA(NO) is not recommended if the procedure attempts to dynamically prepare and execute a DELETE WHERE CURRENT OF statement against an ambiguous cursor after that cursor is opened. You receive a negative SQLCODE if your procedure attempts to use a DELETE WHERE CURRENT OF statement for any of the following cursors:

- A cursor that is using block fetch
- A cursor that is using query parallelism
- A cursor that is positioned on a row that is modified by this or another application process

No is the default.

DEGREE
Specifies whether to attempt to run a query using parallel processing to maximize performance.

1 Specifies that parallel processing should not be used.

1 is the default.

ANY
Specifies that parallel processing can be used.

CONCURRENT ACCESS RESOLUTION
Specifies the whether processing uses only committed data or whether it will wait for commit or rollback of data that is in the process of being updated.

WAIT FOR OUTCOME
Specifies that processing will wait for the commit or rollback of data that is in the process of being updated.

USE CURRENTLY COMMITTED
Specifies that processing use the currently committed version of the data when data that is in the process of being updated is encountered. **USE CURRENTLY COMMITTED** is applicable on scans that access tables that are defined in universal table spaces with row or page level lock size.

When there is lock contention between a read transaction and an insert transaction, **USE CURRENTLY COMMITTED** is applicable to scans with isolation level CS or RS. Applicable scans include intent read scans for read-only and ambiguous queries and for updatable cursors. **USE CURRENTLY COMMITTED** is also applicable to scans initiated from WHERE predicates of UPDATE or DELETE statements and the subselect of INSERT statements.

When there is lock contention is between a read transaction and a delete transaction, **USE CURRENTLY COMMITTED** is applicable to scans with isolation level CS and when CURRENTDATA(NO) is specified.

DYNAMICRULES
Specifies the values that apply, at run time, for the following dynamic SQL attributes:

- The authorization ID that is used to check authorization
- The qualifier that is used for unqualified objects
- The source for application programming options that DB2 uses to parse and semantically verify dynamic SQL statements

DYNAMICRULES also specifies whether dynamic SQL statements can include GRANT, REVOKE, ALTER, CREATE, DROP, and RENAME statements.

In addition to the value of the DYNAMICRULES clause, the run time environment of a native SQL procedure controls how dynamic SQL statements behave at run time. The combination of the DYNAMICRULES value and the run time environment determines the value for the dynamic SQL attributes. That set of attribute values is called the dynamic SQL statement behavior. The following values can be specified:

RUN
Specifies that dynamic SQL statements are to be processed using run behavior.

RUN is the default.

BIND
Specifies that dynamic SQL statements are to be processed using bind behavior.

DEFINEBIND
Specifies that dynamic SQL statements are to be processed using either define behavior or bind behavior.

DEFINERUN
Specifies that dynamic SQL statements are to be processed using either define behavior or run behavior.

INVOKEBIND
Specifies that dynamic SQL statements are to be processed using either invoke behavior or bind behavior.

INVOKERUN
Specifies that dynamic SQL statements are to be processed using either invoke behavior or run behavior.

See "Authorization IDs and dynamic SQL" on page 75 for information on the effects of these options.

APPLICATION ENCODING SCHEME
Specifies the default encoding scheme for SQL variables in static SQL statements in the procedure body. The value is used for defining an SQL variable in a compound statement if the CCSID clause is not specified as part of the data type, and the PARAMETER CCSID routine option is not specified.

ASCII
Specifies that the data is encoded using the ASCII CCSIDs of the server.

EBCDIC
Specifies that the data is encoded using the EBCDIC CCSIDs of the server.

UNICODE
Specifies that the data is encoded using the Unicode CCSIDs of the server.

See the ENCODING bind option in *DB2 Command Reference* for information about how the default for this option is determined.

WITH EXPLAIN or WITHOUT EXPLAIN
Specifies whether information will be provided about how SQL statements in the procedure will execute.

WITHOUT EXPLAIN
Specifies that information will not be provided about how SQL statements in the procedure will execute.

You can get EXPLAIN output for a statement that is embedded in a native SQL procedure that is specified using WITHOUT EXPLAIN by embedding the SQL statement EXPLAIN in the procedure body. Otherwise, the value of the EXPLAIN option applies to all explainable SQL statements in the procedure body, and to the fullselect portion of any DECLARE CURSOR statements.

WITHOUT EXPLAIN is the default.

WITH EXPLAIN
Specifies that information will be provided about how SQL statements in the procedure will execute. Information is inserted into the table *owner*.PLAN_TABLE. *owner* is the authorization ID of the owner of the procedure package. Alternatively, the authorization ID of the owner of the procedure can have an alias as *owner*.PLAN_TABLE that points to the base table, PLAN_TABLE. *owner* must also have the appropriate SELECT and INSERT privileges on that table. WITH EXPLAIN does not obtain information for statements that access remote objects. PLAN_TABLE must have a base table and can have multiple aliases with the same table name, PLAN_TABLE, but have different schema qualifiers; it cannot be a view or a synonym. It should exist before the version is added or replaced. In all inserts to *owner*.PLAN_TABLE, the value of QUERYNO is the statement number that is assigned by DB2.

The WITH EXPLAIN option also populates two optional tables, if they exist: DSN_STATEMNT_TABLE and DSN_FUNCTION_TABLE. DSN_STATEMNT_TABLE contains an estimate of the processing cost for an SQL statement. See *DB2 Application Programming and SQL Guide* for more information. DSN_FUNCTION_TABLE contains information about function resolution. See *DB2 Application Programming and SQL Guide* for more information.

For a description of the tables that are populated by the WITH EXPLAIN option, see "EXPLAIN" on page 1642.

WITH IMMEDIATE WRITE or WITHOUT IMMEDIATE WRITE
Specifies whether immediate writes are to be done for updates that are made to group buffer pool dependent page sets or partitions. This option is only applicable for data sharing environments. The IMMEDWRITE subsystem parameter has no affect of this option. *DB2 Command Reference* shows the implied hierarchy of the IMMEDWRITE bind option (which is similar to this procedure option) as it affects run time.

WITHOUT IMMEDIATE WRITE
Specifies that normal write activity is performed. Updated pages that are group buffer pool dependent are written at or before phase one of commit or at the end of abort for transactions that have been rolled back.

WITHOUT IMMEDIATE WRITE is the default.

WITH IMMEDIATE WRITE
Specifies that updated pages that are group buffer pool dependent are immediately written as soon as the buffer update completes. Updated pages are written immediately even if the buffer is updated during forward progress or during the rollback of a transaction. WITH IMMEDIATE WRITE might impact performance.

ISOLATION LEVEL RR, RS, CS, or UR
Specifies how far to isolate the procedure from the effects of other running applications. For information about isolation levels, see *DB2 Performance Monitoring and Tuning Guide*.

RR Specifies repeatable read.

RS Specifies read stability.

CS Specifies cursor stability. CS is the default.

UR Specifies uncommitted read.

WITH KEEP DYNAMIC or WITHOUT KEEP DYNAMIC
Specifies whether DB2 keeps dynamic SQL statements after commit points.

WITHOUT KEEP DYNAMIC
Specifies that DB2 does not keep dynamic SQL statements after commit points.

WITHOUT KEEP DYNAMIC is the default.

WITH KEEP DYNAMIC
Specifies that DB2 keeps dynamic SQL statements after commit points. If you specify WITH KEEP DYNAMIC, the application does not need to prepare an SQL statement after every commit point. DB2 keeps the dynamic SQL statement until one of the following occurs:

- The application process ends
- A rollback operations occurs
- The application executes an explicit PREPARE statement with the same statement identifier as the dynamic SQL statement

If you specify WITH KEEP DYNAMIC, and the prepared statement cache is active, the DB2 subsystem keeps a copy of the prepared statement in the cache. If the prepared statement cache is not active, the subsystem keeps

only the SQL statement string past a commit point. If the application executes an OPEN, EXECUTE, or DESCRIBE operation for that statement, the statement is implicitly prepared.

If you specify WITH KEEP DYNAMIC, DDF server threads that are used to execute procedures or packages that have this option in effect will remain active. Active DDF server threads are subject to idle thread time outs, as described in *DB2 Installation Guide* for installation panel DSNTIPR.

If you specify WITH KEEP DYNAMIC, you must not specify REOPT ALWAYS. WITH KEEP DYNAMIC and REOPT ALWAYS are mutually exclusive. However, you can specify WITH KEEP DYNAMIC and REOPT ONCE.

Use WITH KEEP DYNAMIC to improve performance if your DRDA client application uses a cursor that is defined as WITH HOLD. The DB2 subsystem automatically closes a held cursor when there are no more rows to retrieve, which eliminates an extra network message.

OPTHINT *string-constant*
Specifies whether query optimization hints are used for static SQL statements that are contained within the body of the procedure.

*string-constant*is a character string of up to 128 bytes in length, which is used by the DB2 subsystem when searching the PLAN_TABLE for rows to use as input. The default value is an empty string, which indicates that the DB2 subsystem does not use optimization hints for static SQL statements.

Optimization hints are only used if optimization hints are enabled for you system. See *DB2 Installation Guide* for information about enabling optimization hints.

SQL PATH
Specifies the SQL path that the DB2 subsystem uses to resolve unqualified user-defined types, functions, and procedure names (in CALL statements) in the body of the procedure.

The maximum length of the SQL path is 2048 bytes. DB2 calculates the length by taking each *schema-name* specified and removing any trailing blanks from it, adding two delimiters around it, and adding one comma after each schema name except for the last one. The length of the resulting string cannot exceed 2048 bytes.

schema-name
Specifies a schema. DB2 does not validate that the specified schema actually exists when the ALTER statement is processed.

| SYSPUBLIC must not be specified for the SQL path.

schema-name-list
Specifies a comma separated list of schema names. The same schema name should not appear more than one time in the list of schema names. The number of schema names that you can specify is limited by the maximum length of the resulting SQL path.

| SYSPUBLIC must not be specified for the SQL path.

SYSTEM PATH
Specifies the schema names "SYSIBM", "SYSFUN", "SYSPROC", "SYSIBMADM".

SESSION_USER or USER
Specifies the value of the SESSION_USER (or USER) special register. At the

time the ALTER statement is processed, the actual length is included in the total length of the list of schema names that is specified for the PATH option. If you specify SESSION_USER (or USER) in a list of schema names, do not use delimiters around the SESSION_USER (or USER) keyword.

RELEASE AT
Specifies when to release resources that the procedure uses: either at each commit point or when the procedure terminates.

COMMIT
Specifies that resources will be released at each commit point.

COMMIT is the default.

DEALLOCATE
Specifies that resources will be released only when the procedure terminates. DEALLOCATE has no effect on packages that run on a DB2 server through a DRDA connection with a client system. DEALLOCATE also has no effect on dynamic SQL statements, which always use RELEASE AT COMMIT, with this exception: When you use the RELEASE AT DEALLOCATE clause and the WITH KEEP DYNAMIC clause, and the subsystem is installed with a value of YES for the field CACHE DYNAMIC SQL on installation panel DSNTIP8, the RELEASE AT DEALLOCATE option is honored for dynamic SELECT and SQL data change statements.

Locks that are acquired for dynamic statements are held unit one of the following events occurs:

- The application process ends.
- The application process issues a PREPARE statement with the same statement identifier. (Locks are released at the next commit point).
- The statement is removed from the prepared statement cache because the statement has not been used. (Locks are released at the next commit point).
- An object that the statement is dependent on is dropped or altered, or a privilege that the statement needs is revoked. (Locks are released at the next commit point).

RELEASE AT DEALLOCATE can increase the package size because additional items become resident in the package. For more information about how the RELEASE clause affects locking and concurrency, see *DB2 Performance Monitoring and Tuning Guide*.

REOPT
Specifies if DB2 will determine the access path at run time by using the values of SQL variables or SQL parameters, parameter markers, and special registers.

NONE
Specifies that DB2 does not determine the access path at run time by using the values of SQL variables or SQL parameters, parameter markers, and special registers.

NONE is the default.

ALWAYS
Specifies that DB2 always determine the access path at run time each time an SQL statement is run.

ONCE
Specifies that DB2 determine the access path for any dynamic SQL statements only one time, at the first time the statement is opened. This

access path is used until the prepared statement is invalidated or removed from the dynamic statement cache and needs to be prepared again.

VALIDATE RUN or VALIDATE BIND
Specifies whether to recheck, at run time, errors of the type "OBJECT not FOUND" and NOT AUTHORIZED" that are found during bind or rebind. The option has no effect if all objects and needed privileges exist.

VALIDATE RUN
Specifies that if needed objects or privileges do not exist when the ALTER PROCEDURE statement is processed, warning messages are returned, but the ALTER PROCEDURE statement succeeds. The DB2 subsystem rechecks for the objects and privileges at run time for those SQL statements that failed the checks during processing of the ALTER PROCEDURE statement. The authorization checks the use of the authorization ID of the owner of the procedure package.

VALIDATE RUN is the default.

VALIDATE BIND
Specifies that if needed objects or privileges do not exist at the time the ALTER PROCEDURE statement is processed, an error is issued and the ALTER PROCEDURE statement fails.

ROUNDING
Specifies the rounding mode for manipulation of DECFLOAT data.

DEC_ROUND_CEILING
Specifies numbers are rounded towards positive infinity.

DEC_ROUND_DOWN
Specifies numbers are rounded towards 0 (truncation).

DEC_ROUND_FLOOR
Specifies numbers are rounded towards negative infinity.

DEC_ROUND_HALF_DOWN
Specifies numbers are rounded to nearest; if equidistant, round down.

DEC_ROUND_HALF_EVEN
Specifies numbers are rounded to nearest; if equidistant, round so that the final digit is even.

DEC_ROUND_HALF_UP
Specifies numbers are rounded to nearest; if equidistant, round up.

DEC_ROUND_UP
Specifies numbers are rounded away from 0.

DATE FORMAT ISO, EUR, USA, JIS, or LOCAL
Specifies the date format for result values that are string representations of date or time values. See “String representations of datetime values” on page 101 for more information.

The default format is specified in the DATE FORMAT field of installation panel DSNTIP4 of the system where the procedure is defined. You cannot use the LOCAL option unless you have a date exit routine.

DECIMAL(15), DECIMAL(31), DECIMAL(15,*s*), or DECIMAL(31,*s*)
Specifies the maximum precision that is to be used for decimal arithmetic operations. See “Arithmetic with two decimal operands” on page 244 for more information. The default format is specified in the DECIMAL ARITHMETIC field of installation panel DSNTIPF of the system where the procedure is

defined. If the form *pp.s* is specified, *s* must be a number between 1 and 9. *s* represents the minimum scale that is to be used for division.

FOR UPDATE CLAUSE OPTIONAL or FOR UPDATE CLAUSE REQUIRED
Specifies whether the FOR UPDATE clause is required for a DECLARE CURSOR statement if the cursor is to be used to perform positioned updates.

FOR UPDATE CLAUSE REQUIRED
Specifies that a FOR UPDATE clause must be specified as part of the cursor definition if the cursor will be used to make positioned updates.

FOR UPDATE CLAUSE REQUIRED is the default.

FOR UPDATE CLAUSE OPTIONAL
Specifies that the FOR UPDATE clause does not need to be specified in order for a cursor to be used for positioned updates. The procedure body can include positioned UPDATE statements that update columns that the user is authorized to update.

If the resulting DBRM for the procedure is very large, you might need extra storage when you specify FOR UPDATE CLAUSE OPTIONAL.

The FOR UPDATE clause of the select-statement with no column list applies to static or dynamic SQL statements. You can specify the FOR UPDATE OF clause of the select-statement with a column list to restrict updates to only the columns that are named in the column list and to specify the acquisition of update locks.

TIME FORMAT ISO, EUR, USA, JIS, or LOCAL
Specifies the time format for result values that are string representations of date or time values. See "String representations of datetime values" on page 101 for more information.

The default format is specified in the TIME FORMAT field of installation panel DSNTIP4 of the system where the procedure is defined. You cannot use the LOCAL option unless you have a date exit routine.

BUSINESS_TIME SENSITIVE
Determines whether references to application-period temporal tables in both static and dynamic SQL statements are affected by the value of the CURRENT TEMPORAL BUSINESS_TIME special register.

YES
References to application-period temporal tables are affected by the value of the CURRENT TEMPORAL BUSINESS_TIME special register. YES is the default value.

NO References to application-period temporal tables are not affected by the value of the CURRENT TEMPORAL BUSINESS_TIME special register.

Related information:

"CURRENT TEMPORAL BUSINESS_TIME" on page 194

SYSTEM_TIME SENSITIVE
Determines whether references to system-period temporal tables in both static and dynamic SQL statements are affected by the value of the CURRENT TEMPORAL SYSTEM_TIME special register.

YES
References to system-period temporal tables are affected by the value of the CURRENT TEMPORAL SYSTEM_TIME special register. YES is the default value.

NO References to system-period temporal tables are not affected by the value of the CURRENT TEMPORAL SYSTEM_TIME special register.

Related information:

"CURRENT TEMPORAL SYSTEM_TIME" on page 196

ARCHIVE SENSITIVE

Determines whether references to archive-enabled tables in SQL statements are affected by the value of the SYSIBMADM.GET_ARCHIVE global variable.

YES

References to archive-enabled tables are affected by the value of the SYSIBMADM.GET_ARCHIVE global variable. YES is the default value.

NO References to archive-enabled tables are not affected by the value of the SYSIBMADM.GET_ARCHIVE global variable.

Related information:

"References to built-in global variables" on page 223

APPLCOMPAT *compatibility-level*

Specifies the package compatibility level behavior for static SQL, If this option is not specified then the behavior is determined, in priority order, by the *compatibility-level* of the last BIND or REBIND of the package or the APPLCOMPAT system parameter. The following values of *compatibility-level* can be specified:

V10R1

The static SQL statements in the package have V10R1 compatibility behavior.

V11R1

The static SQL statements in the package have V11R1 compatibility behavior.

Related information:

APPL COMPAT LEVEL field (APPLCOMPAT subsystem parameter) (DB2 Installation and Migration)

SQL-routine-body

Specifies the statements that define the body of the SQL procedure. For information on the SQL control statements that are supported in native SQL procedures, see Chapter 6, "SQL control statements for SQL routines," on page 1963. If an *SQL-procedure-statement* is the only statement in the procedure body, the statement must not end with a semicolon.

Notes

Considerations for altering a version of a procedure: To alter a version of a procedure, the environment settings that are in effect when the ALTER PROCEDURE statement is issued must be the same as the environment settings that are in effect when the version of the procedure is first created using the CREATE PROCEDURE or ALTER PROCEDURE statements if one of the following options is specified:

- **QUALIFIER**
- **PACKAGE OWNER**
- **WLM ENVIRONMENT FOR DEBUG MODE**
- **OPTHINT**

- **SQL PATH**
- **DECIMAL** (if the value includes a comma

Changing to a native SQL procedure: You cannot change an external SQL procedure to a native SQL procedure. You can drop the external SQL procedure that you want to change by using the DROP statement and create a native SQL procedure with a similar definition using the CREATE PROCEDURE statement. Alternatively, you can create a native SQL procedure using a different schema.

Identifier resolution: See Chapter 6, "SQL control statements for SQL routines," on page 1963 for information on how names are resolved to columns, SQL variables, or SQL routines for native SQL procedures. Name resolution is unchanged for external SQL procedures.

If duplicate names are used for columns and SQL variables and parameters, qualify the duplicate names by using the table designator for columns, the procedure name for parameters, and the label name for SQL variables.

Characteristics of the package that is generated for a version of a procedure: The package that is associated with a version of a procedure is named as follows:

- *location* is set to the value of the CURRENT SERVER special register
- *collection-id* (schema) for the package is the same as the schema qualifier of the procedure
- *package-id* is the same as the specific name of the procedure
- *version-id* is the same as the version identifier for the initial version of the procedure

The package is generated using the bind options that correspond to the implicitly or explicitly specified procedure options. See Table 98 on page 969 for more information. In addition to the corresponding bind options, the package is generated using the following bind options:

- DBPROTOCAL(DRDA)
- FLAG(1)
- SQLERROR(NOPACKAGE)
- ENABLE(*)

Considerations for a procedure that is defined using a TABLE LIKE name AS LOCATOR clause: If a procedure is defined with a table parameter (the **TABLE LIKE** *name* **AS LOCATOR** clause was specified in the CREATE PROCEDURE statement to indicate that one of the input parameters is a transition table), the procedure cannot be changed with an ALTER PROCERDURE statement if the change requires that the parameter list be specified. For example, to add or replace a version of a native SQL procedure, the procedure must be dropped and re-created.

Considerations for SQL processor programs: SQL processor programs, such as SPUFI, the command line processor, and DSNTEP2, might not correctly parse SQL statements in the routine body that end with semicolons. These processor programs accept multiple SQL statements as input, with each statement separated with a terminator character. Processor programs that use a semicolon as the SQL statement terminator can truncate a CREATE FUNCTION statement with embedded semicolons and pass only a portion of it to DB2. Therefore, you might need to change the SQL terminator character for these processor programs. For

information on changing the terminator character for SPUFI and DSNTEP2, see *DB2 Application Programming and SQL Guide*.

Correspondence of procedure options to BIND options: The following table lists options for CREATE PROCEDURE and ALTER PROCEDURE and the corresponding options for the bind commands. See *DB2 Command Reference* for more information about the effects of the options of the bind commands.

Table 98. Correspondence of procedure options to bind options

CREATE PROCEDURE or ALTER PROCEDURE option	bind commands option
APPLICATION ENCODING SCHEME	ENCODING(ASCII) ENCODING(EBCDIC) ENCODING(UNICODE)
ARCHIVE SENSITIVE NO	ARCHIVESENSITIVE(NO)
ARCHIVE SENSITIVE YES	ARCHIVESENSITIVE(YES)
BUSINESS_TIME SENSITIVE NO	BUSTIMESENSITIVE(NO)
BUSINESS_TIME SENSITIVE YES	BUSTIMESENSITIVE(YES)
CURRENT DATA NO	CURRENTDATA(NO)
CURRENT DATA YES	CURRENTDATA(YES)
DEFER PREPARE	DEFER(PREPARE)
NODEFER PREPARE	NODEFER(PREPARE)
DEGREE	DEGREE(ANY) DEGREE(1)
DYNAMICRULES	DYNAMICRULES(RUN) DYNAMICRULES(BIND) DYNAMICRULES(DEFINEBIND) DYNAMICRULES(DEFINERUN) DYNAMICRULES(INVOKEBIND) DYNAMICRULES(INVOKERUN)
ISOLATION LEVEL	ISOLATION(RR) ISOLATION(RS) ISOLATION(CS) ISOLATION(UR)
OPTHINT	OPTHINT
PACKAGE OWNER	OWNER
QUALIFIER	QUALIFIER
RELEASE AT COMMIT	RELEASE(COMMIT)
RELEASE AT DEALLOCATE	RELEASE(DEALLOCATE)
REOPT ALWAYS	REOPT(ALWAYS)
REOPT NONE	REOPT(NONE)
REOPT ONCE	REOPT(ONCE)
ROUNDING DEC_ROUND_CEILING	ROUNDING(CEILING)
ROUNDING DEC_ROUND_DOWN	ROUNDING(DOWN)

Table 98. Correspondence of procedure options to bind options (continued)

CREATE PROCEDURE or ALTER PROCEDURE option	bind commands option
ROUNDING DEC_ROUNDING_FLOOR	ROUNDING(FLOOR)
ROUNDING DEC_ROUNDING_HALF_DOWN	ROUNDING(HALFDOWN)
ROUNDING DEC_ROUNDING_HALF_EVEN	ROUNDING(HALFEVEN)
ROUNDING DEC_ROUNDING_HALF_UP	ROUNDING(HALFUP)
ROUNDING DEC_ROUNDING_UP	ROUNDING(UP)
SQL PATH	PATH
SYSTEM_TIME SENSITIVE NO	SYSTIMESENSITIVE(NO)
SYSTEM_TIME SENSITIVE YES	SYSTIMESENSITIVE(YES)
VALIDATE BIND	VALIDATE(BIND)
VALIDATE RUN	VALIDATE(RUN)
WITH EXPLAIN	EXPLAIN(YES)
WITHOUT EXPLAIN	EXPLAIN(NO)
WITH IMMEDIATE WRITE	IMMEDWRITE(YES)
WITHOUT IMMEDIATE WRITE	IMMEDWRITE(NO)
WITH KEEPDYNAMIC	KEEPDYNAMIC(YES)
WITHOUT KEEPDYNAMIC	KEEPDYNAMIC(NO)

Invalidation of packages: When a version of an SQL procedure is altered to change any option that is specified for the active version, all the packages that refer to that procedure are marked invalid. Additionally, when certain attributes of a native SQL procedure are changed, the body of the procedure might be rebound or regenerated. The following table summarizes when implicit rebind and regeneration occurs when specific options are changed. A value of Y in a row indicates that a rebind or regeneration occurs if the option is changed for a version of the procedure. A value of N in a row indicates that a rebind or regeneration does not occur.

Table 99. CREATE PROCEDURE and ALTER PROCEDURE options that result in rebind or regeneration when changed.

CREATE PROCEDURE or ALTER PROCEDURE option	Change requires rebind of invoking applications?	Change results in implicit rebind of the non-control statements of the body of the procedure?	Change results in implicit regeneration of the entire body of the procedure?
ALLOW DEBUG MODE, DISALLOW DEBUG MODE, or DISABLE DEBUG MODE	Y[1, 2]	Y[1]	Y
APPLICATION ENCODING SCHEME	Y	Y	Y
ARCHIVE SENSITIVE	Y	Y	Y
ASUTIME	Y	N	N

Table 99. CREATE PROCEDURE and ALTER PROCEDURE options that result in rebind or regeneration when changed. (continued)

CREATE PROCEDURE or ALTER PROCEDURE option	Change requires rebind of invoking applications?	Change results in implicit rebind of the non-control statements of the body of the procedure?	Change results in implicit regeneration of the entire body of the procedure?
BUSINESS_TIME SENSITIVE	Y	Y	Y
COMMIT ON RETURN	Y	N	N
CURRENT DATA	N	Y	N
DATE FORMAT	Y	Y	Y
DECIMAL	Y	Y	Y
DEFER PREPARE or NODEFER PREPARE	N	Y	N
DEGREE	N	Y	N
DYNAMIC RESULT SETS	Y	N	N
DYNAMICRULES	N	Y	N
FOR UPDATE CLAUSE OPTIONAL or FOR UPDATE CLAUSE REQUIRED	Y	Y	Y
INHERIT SPECIAL REGISTERS or DEFAULT SPECIAL REGISTERS	Y	N	N
ISOLATION LEVEL	N	Y	N
MODIFIES SQL DATA, READS SQL DATA, or CONTAINS SQL	Y	Y	Y
NOT DETERMINISTIC or DETERMINISTIC	N	N	N
OPTHINT	N	Y	N
PACKAGE OWNER	N	Y	N
QUALIFIER	N	Y	N
RELEASE AT COMMIT or RELEASE AT DEALLOCATE	N	Y	N
REOPT	N	Y	N
SQL PATH	N	Y	N
STOP AFTER SYSTEM DEFAULT FAILURES, STOP AFTER nn FAILURES, or CONTINUE AFTER FAILURES	Y	N	N
SYSTEM_TIME SENSITIVE	Y	Y	Y
TIME FORMAT	Y	Y	Y
VALIDATE RUN or VALIDATE BIND	N	Y	N
WITH EXPLAIN or WITHOUT EXPLAIN	N	Y	N

Table 99. CREATE PROCEDURE and ALTER PROCEDURE options that result in rebind or regeneration when changed. (continued)

CREATE PROCEDURE or ALTER PROCEDURE option	Change requires rebind of invoking applications?	Change results in implicit rebind of the non-control statements of the body of the procedure?	Change results in implicit regeneration of the entire body of the procedure?
WITH IMMEDIATE WRITE or WITHOUT IMMEDIATE WRITE	N	Y	N
WITH KEEP DYNAMIC or WITHOUT KEEP DYNAMIC	N	Y	N
WLM ENVIRONMENT FOR DEBUG MODE	Y	N	N

Note:

1. The procedure package is rebound or regenerated if a value of ALLOW DEBUG MODE is changed to DISALLOW DEBUG MODE.
2. Invoking applications are invalidated if a value of DISALLOW DEBUG MODE is changed to DISABLE DEBUG MODE.

Considerations for SYSENVIRONMENTS catalog table: An ALTER statement that specifies a new environment settings will result in a new row being added to the SYSENVIRONMENTS catalog table. The new row will be added even if an error is subsequently encountered during processing of the ALTER statement. Thus, a new SYSENVIRONMENTS row might be added even for an ALTER statement that fails.

Stored procedures with a parameter that is defined as an array type: A procedure that is defined with a parameter that is an array type can be invoked only from within an SQL PL context.

Compatibilities: For compatibility with previous versions of DB2, the following clauses can be specified, but they will be ignored and an error or a warning will be issued. If ALTER is implicitly or explicitly specified with one of these options specified as part of *option-list*, a warning is issued and the option is ignored. If REPLACE or ADD VERSION is specified with one of these options specified as part of *option-list*, and error is issued. For example, ADD VERSION is specified and STAY RESIDENT is specified as part of option-list and error is issued.

- STAY RESIDENT
- PROGRAM TYPE
- RUN OPTIONS
- NO DBINFO
- COLLID or NOCOLLID
- SECURITY
- PARAMETER STYLE GENERAL WITH NULLS
- STOP AFTER SYSTEM DEFAULT FAILURES
- STOP AFTER *nn* FAILURES
- CONTINUE AFTER FAILURES

If WLM ENVIRONMENT is specified for a native SQL procedure, WLM ENVIRONMENT FOR DEBUG MODE must be specified.

For compatibility with the CREATE PROCEDURE statement, the following clause can be specified, but will be ignored:

- **LANGUAGE SQL**

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords:

- RESULT SET, RESULT SETS, and DYNAMIC RESULT SET as synonyms for **DYNAMIC RESULT SETS**.
- VARIANT as a synonym for **NOT DETERMINISTIC**
- NOT VARIANT as a synonym for **DETERMINISTIC**

Considerations for catalog comments for a routine definition: When a function definition is replaced any existing comment in the catalog for the definition is removed, and when a function definition is regenerated any existing comment in the catalog for the definition is retained.

Example

Example 1: The following statement changes the existing procedure options for the active version of the UPDATE_SALARY_1 native SQL procedure. If you need to change a different version of the procedure, you would specify `VERSION routine-version-id` in place of ACTIVE VERSION. Note that the ALTER clause that precedes the version specification can be omitted.

```
ALTER PROCEDURE UPDATE_SALARY_1
      ALTER ACTIVE VERSION
      NOT DETERMINISTIC
      CALLED ON NULL INPUT
      ALLOW DEBUG MODE
      ASUTIME LIMIT 10
```

Example 2: To change the procedure body of any existing version of a procedure, you need to use the REPLACE clause. The following statement changes both the procedure body and the existing SQL data access option for version V2 of the UPDATE_SALARY_1 SQL procedure. Note that the list of parameters is specified even though no changes are made to the list. To replace an existing version of a procedure, you must specify the list of parameters, any options that are to have non-default values (even if those options are specified in the version of the procedure that you are replacing), and the body of the procedure.

```
ALTER PROCEDURE UPDATE_SALARY_1
      REPLACE VERSION V2 (P1 INTEGER, P2 CHAR(5))
      MODIFIES SQL DATA
      UPDATE EMP SET SALARY = SALARY * RATE
            WHERE EMPNO = EMPLOYEE_NUMBER;
```

Example 3: To add a new version of an existing procedure, use the ADD VERSION clause. The following statement adds a new version of the UPDATE_SALARY_1 procedure to apply a larger salary increase. Note that the list of parameters is specified even though the new version of the procedure uses the same parameters as the existing version of the procedure. To add a new version of a procedure, you must specify the list of parameters, any options that will have non-default values, and the body of the procedure.

```
ALTER PROCEDURE UPDATE_SALARY_1
      ADD VERSION V3 (P1 INTEGER, P2 CHAR(5))
        UPDATE EMP SET SALARY = SALARY * (RATE*10)
            WHERE EMPNO = EMPLOYEE_NUMBER;
```

Example 4: When the new version of the procedure has been defined, as in Example 3, you must use the ALTER PROCEDURE statement with the ACTIVATE VERSION clause if the new version of the procedure is to be the currently active version, as in the following example.

```
ALTER PROCEDURE UPDATE_SALARY_1
      ACTIVATE VERSION V3;
```

Example 5: To regenerate the currently active version of a procedure, use the following statement.

```
ALTER PROCEDURE UPDATE_SALARY_1
      REGENERATE ACTIVE VERSION;
```

ALTER SEQUENCE

The ALTER SEQUENCE statement changes the attributes of a sequence at the current server. Only future values of the sequence are affected by the ALTER SEQUENCE statement.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

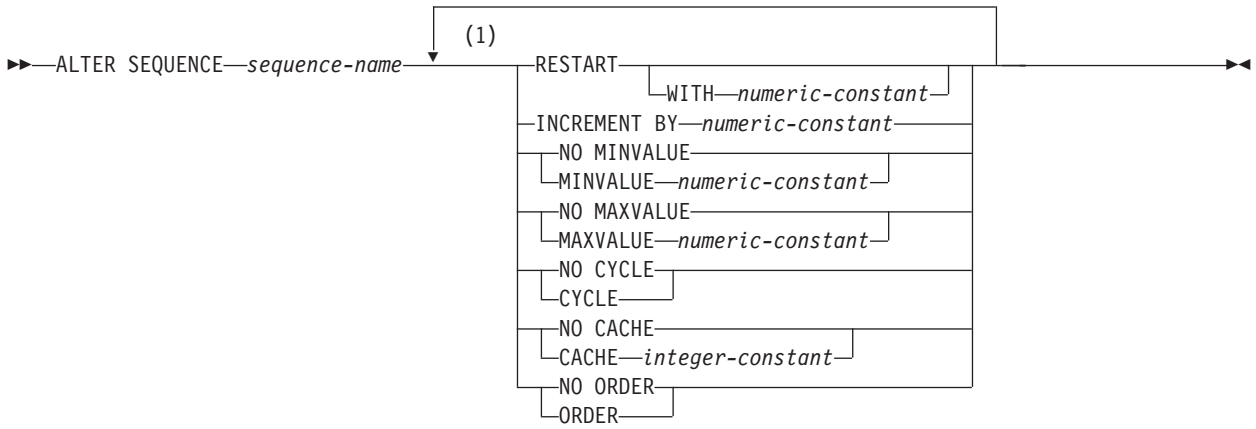
The privilege set that is defined below must include at least one of the following:
- Ownership of the sequence
- The ALTER privilege for the sequence
- The ALTERIN privilege on the schema
- SYSADM or SYSCTRL authority
- System DBADM

Installation SYSADM privilege is required to alter the SYSIBM.DSNSEQ_IMPLICITDB sequence (which specifies the maximum number of implicitly created databases).

The authorization ID that matches the schema name implicitly has the ALTERIN privilege on the schema.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the package. If the statement is dynamically prepared, the privilege set is the union of the privilege sets that are held by each authorization ID and role of the process.

Syntax



Notes:

1 At least one option must be specified and the same clause must not be specified more than once. Separator commas can be specified between sequence attributes when a sequence is defined.

Description

sequence-name
Identifies the sequence. The combination of sequence name and the implicit or explicit qualifier must identify an existing sequence at the current server. *sequence-name* must not identify a sequence that is generated by DB2 for an identity column or a DB2_GENERATED_DOCID_FOR_XML column.

RESTART
Restarts the sequence. If *numeric-constant* is not specified, the sequence is restarted at the value specified implicitly or explicitly as the starting value on the CREATE SEQUENCE statement that originally created the sequence.

WITH *numeric-constant*
Specifies the value at which to restart the sequence. The value can be any positive or negative value that could be assigned to the a column of the data type that is associated with the sequence without non-zero digits existing to the right of the decimal point.

If RESTART is not specified, the sequence is not restarted. Instead, it resumes with the current values in effect for all the options after the ALTER statement is issued.

After a sequence is restarted or changed to allow cycling, sequence numbers might be duplicates of values generated by the sequence previously.

INCREMENT BY *numeric-constant*
Specifies the interval between consecutive values of the sequence. The value can be any positive or negative value (including 0) that could be assigned to a column of the data type that is associated with the sequence without any non-zero digits existing to the right of the decimal point.

If INCREMENT BY *numeric-constant* is positive, the sequence ascends. If INCREMENT BY *numeric-constant* is negative, the sequence descends. If INCREMENT BY *numeric-constant* is 0, the sequence is treated as an ascending sequence.

The absolute value of INCREMENT BY can be greater than the difference between MAXVALUE and MINVALUE.

NO MINVALUE or MINVALUE
Specifies whether or not there is a minimum end point of the range of values for the sequence.

NO MINVALUE
Specifies that the minimum end point of the range of values for the sequence has not been specified explicitly. In such a case, the value for MINVALUE becomes one of the following:

- For an ascending sequence, the value is the original starting value.
- For a descending sequence, the value is the minimum of the data type that is associated with the sequence.

MINVALUE *numeric-constant*
Specifies the minimum value at which a descending sequence either cycles or stops generating values, or an ascending sequence cycles to after reaching the maximum value. The last value that is generated for a cycle of a descending sequence will be equal to or greater than this value. MINVALUE is the value to which an ascending sequence cycles to after reaching the maximum value.

The value can be any positive or negative value that could be assigned to the a column of the data type that is associated with the sequence without non-zero digits existing to the right of the decimal point. The value must be less than or equal to the maximum value.

NO MAXVALUE or MAXVALUE
Specifies whether or not there is a maximum end point of the range of values for the sequence.

NO MAXVALUE
Specifies either explicitly or implicitly that the minimum end point of the range of values for the sequence has not be set. In such a case, the default value for MAXVALUE becomes one of the following:

- For an ascending sequence, the value is the maximum value of the data type that is associated with the sequence
- For a descending sequence, the value is the original starting value.

If NO MAXVALUE is explicitly specified in the ALTER SEQUENCE statement, the value of the MAXVALUE column in the catalog table is reset to the maximum value of the data type associated with the sequence if the sequence is ascending or the value stored in the START column of the catalog table if the sequence is descending. Whether the sequence is ascending or descending depends on whether or not the INCREMENT BY option is reset. If it is, the new INCREMENT BY VALUE determines if the sequence is ascending or descending. If it is not explicitly reset, the value stored in the INCREMENT column of the catalog table determines if the sequence is ascending or descending.

MAXVALUE *numeric-constant*
Specifies the maximum value at which an ascending sequence either cycles or stops generating values or a descending sequence cycles to after

reaching the minimum value. The last value that is generated for a cycle of an ascending sequence will be less than or equal to this value. MAXVALUE is the value to which a descending sequence cycles to after reaching the minimum value.

The value can be any positive or negative value that could be assigned to the a column of the data type that is associated with the sequence without non-zero digits existing to the right of the decimal point. The value must be greater than or equal to the minimum value.

NO CYCLE or CYCLE
Specifies whether or not the sequence should continue to generate values after reaching either its maximum or minimum value. The boundary of the sequence can be reached either with the next value landing exactly on the boundary condition or by overshooting it.

NO CYCLE
Specifies that the sequence cannot generate more values once the maximum or minimum value for the sequence has been reached.

CYCLE
Specifies that the sequence continue to generate values after either the maximum or minimum value has been reached. If this option is used, after an ascending sequence reaches its maximum value, it generates its minimum value. After a descending sequence reaches its minimum value, it generates its maximum value. The maximum and minimum values for the sequence defined by the MINVALUE and MAXVALUE options determine the range that is used for cycling.

When CYCLE is in effect, duplicate values can be generated by the sequence. When a sequence is defined with CYCLE, any application conversion tools for converting applications from other vendor platforms to DB2 should also explicitly specify MINVALUE, MAXVALUE, and START WITH values.

NO CACHE or CACHE
Specifies whether or not to keep some preallocated values in memory for faster access. This is a performance and tuning option.

NO CACHE
Specifies that values of the sequence are not to be preallocated. This option ensures that there is not a loss of values in the case of a system failure. When NO CACHE is specified, the values of the sequence are not stored in the cache. In this case, every request for a new value for the sequence results in synchronous I/O.

CACHE *integer-constant*
Specifies the maximum number of sequence values that DB2 can preallocate and keep in memory. Preallocating values in the cache reduces synchronous I/O when values are generated for the sequence. The actual number of values that DB2 caches is always the lesser of the number in effect for the CACHE option and the number of remaining values within the logical range. Thus, the CACHE value is essentially an upper limit for the size of the cache.

In the event the system is shut down (either normally or through a system failure), all cached sequence values that have not been used in committed statements are lost (that is, they will never be used). The value specified for the CACHE option is the maximum number of sequence values that could be lost when the system is shut down.

The minimum value is 2.

In a data sharing environment, you can use the CACHE and NO ORDER options to allow multiple DB2 members to cache sequence values simultaneously.

NO ORDER or ORDER
Specifies whether the sequence numbers must be generated in order of request.

NO ORDER
Specifies that the sequence numbers do not need to be generated in order of request.

ORDER
Specifies that the sequence numbers are generated in order of request. Specifying ORDER might disable the caching of values. There is no guarantee that values are assigned in order across the entire server unless NO CACHE is also specified. ORDER applies only to a single-application process.

In a data sharing environment, if the CACHE and NO ORDER options are in effect, multiple caches can be active simultaneously, and the requests for next value assignments from different DB2 members might not result in the assignment of values in strict numeric order. For example, if members DB2A and DB2B are using the same sequence, and DB2A gets the cache values 1 to 20 and DB2B gets the cache values 21 to 40, the actual order of values assigned would be 1,21,2 if DB2A requested for next value first, then DB2B requested, and then DB2A again requested. Therefore, to guarantee that sequence numbers are generated in strict numeric order among multiple DB2 members using the same sequence concurrently, specify the ORDER option.

Notes

Altering a sequence: The changes to the attributes of a sequence take effect after the ALTER SEQUENCE statement is committed. Only future sequence numbers are affected by the ALTER SEQUENCE statement. If the ALTER SEQUENCE request results in an error or is rolled back, nothing is changed; however, unused cache values might be lost.

- The data type of a sequence cannot be changed. Instead, drop and re-create the sequence specifying the desired data type for the new sequence.
- All cached values are lost when a sequence is altered.
- After restarting a sequence or changing it to cycle, it is possible that a generated value will duplicate a value previously generated for that sequence.

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords:

- NOCACHE (single key word) as a synonym for NO CACHE
- NOCYCLE (single key word) as a synonym for NO CYCLE
- NOMINVALUE (single key word) as a synonym for NO MINVALUE
- NOMAXVALUE (single key word) as a synonym for NO MAXVALUE
- NOORDER (single key word) as a synonym for NO ORDER

Examples

Example 1: Reset a sequence to the START WITH value to generate the numbers from 1 up to the number of rows in the table:

```
ALTER SEQUENCE org_seq
  RESTART;
```

ALTER STOGROUP

The ALTER STOGROUP statement changes the description of a storage group at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

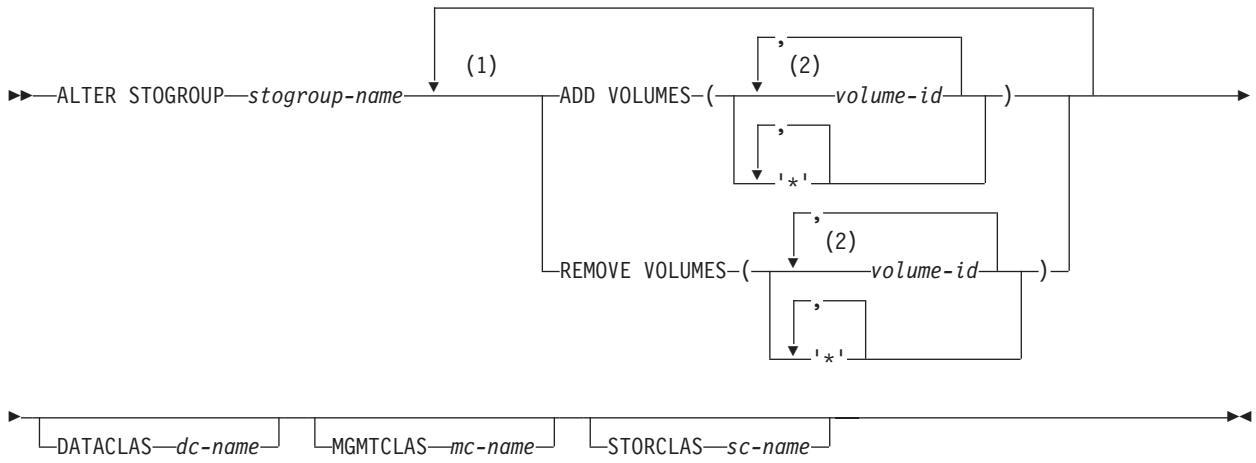
Authorization

The privilege set that is defined below must include one of the following:

- Ownership of the storage group
- SYSADM or SYSCTRL authority

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the package. If the statement is dynamically prepared, the privilege set is the union of the privilege sets that are held by each authorization ID and role of the process.

Syntax



Notes:

1 The same clause must not be specified more than once.

2 The same *volume-id* must not be specified more than once in the same clause.

Description

stogroup-name
Identifies the storage group to be altered. The name must identify a storage group that exists at the current server.

ADD VOLUMES(*volume-id*,...) or ADD VOLUMES('*',...)
Adds volumes to the storage group. Each *volume-id* is the volume serial number of a storage volume to be added. It can have a maximum of six characters and is specified as an identifier or a string constant.

A *volume-id* must not be specified if any volume of the storage group is designated by an asterisk (*). An asterisk must not be specified if any volume of the storage group is designated by a *volume-id*.

You cannot add a volume that is already in the storage group unless you first remove it with REMOVE VOLUMES.

Asterisks are recognized only by Storage Management Subsystem (SMS). If the data set that is associated with the storage group is non SMS managed, either ADD VOLUMES or REMOVE VOLUMES must be specified. Neither ADD VOLUMES or REMOVE VOLUMES is required if DATACLAS, MGMTCLAS, or STORCLAS is specified. SMS usage is recommended, rather than using DB2 to allocate data to specific volumes. Having DB2 select the volume requires non-SMS usage or assigning an SMS Storage Class with guaranteed space. However, because guaranteed space reduces the benefits of SMS allocation, it is not recommended.

If you do choose to use specific volume assignments, additional manual space management must be performed. Free space must be managed for each individual volume to prevent failures during the initial allocation and extension. This process generally requires more time for space management and results in more space shortages. Guaranteed space should be used only where the space needs are relatively small and do not change.

REMOVE VOLUMES(*volume-id*,...) or REMOVE VOLUMES('*',...)
Removes volumes from the storage group. Each *volume-id* is the volume serial number of a storage volume to be removed. Each *volume-id* must identify a volume that is in the storage group.

The REMOVE VOLUMES clause is applied to the current list of volumes before the ADD VOLUMES clause is applied. Removing a volume from a storage group does not affect existing data, but a volume that has been removed is not used again when the storage group is used to allocate storage for table spaces or index spaces.

Asterisks are recognized only by Storage Management Subsystem (SMS). If the data set that is associated with the storage group is non SMS managed, either ADD VOLUMES or REMOVE VOLUMES must be specified. Neither ADD VOLUMES or REMOVE VOLUMES is required if DATACLAS, MGMTCLAS, or STORCLAS is specified.

DATACLAS *dc-name*
Identifies the name of the SMS data class to associate with the DB2 storage group. The SMS data class name must be from 1-8 characters in length. The SMS storage administrator defines the data class that can be used. DATACLAS must not be specified more than one time.

MGMTCLAS *mc-name*
Identifies the name of the SMS management class to associate with the DB2 storage group. The SMS management class name must be from 1-8 characters in length. The SMS storage administrator defines the management class that can be used. MGMTCLAS must not be specified more than one time.

STORCLAS *sc-name*
Identifies the name of the SMS storage class to associate with the DB2 storage group. The SMS storage class name must be from 1-8 characters in length. The

SMS storage administrator defines the storage class that can be used. STORCLAS must not be specified more than one time.

Notes

Work file databases: If the storage group altered contains data sets in a work file database, the database must be stopped and restarted for the effects of the ALTER to be recognized. To stop and restart a database, issue the following commands:

```
-STOP DATABASE(database-name)
-START DATABASE(database-name)
```

Device types: When the storage group is used at run time, an error can occur if the volumes in the storage group are of different device types, or if a volume is not available to z/OS for dynamic allocation of data sets.

When a storage group is used to extend a data set, all volumes in the storage group must be of the same device type as the volumes used when the data set was defined. Otherwise, an extend failure occurs if an attempt is made to extend the data set.

Number of volumes: There is no specific limit on the number of volumes that can be defined for a storage group. However, the maximum number of volumes that can be managed for a storage group is 133.

If the VOLUMES clause is specified, the maximum number of volumes is 59.

Verifying the existence of volumes and classes: When processing the VOLUMES, DATACLAS, MGMTCLAS, or STORCLAS clauses, DB2 does not check the existence of the volumes or classes or determine the types of devices that are identified or if SMS is active. Later, when the storage group allocates data sets, the list of volumes is passed in the specified order to Data Facilities (DFSMSdfp). See *DB2 Administration Guide* for more information about creating DB2 storage groups.

SMS data set management: You can have Storage Management Subsystem (SMS) manage the storage needed for the objects that the storage group supports. To do so, specify ADD VOLUMES('*') and REMOVE VOLUMES(*current-vols*) in the ALTER statement, where *current-vols* is the list of the volumes currently assigned to the storage group. SMS manages every data set created later for the storage group. SMS does not manage data sets created before the execution of the statement.

You can also specify ADD VOLUMES(*volume-id*) and REMOVE VOLUMES('*') to make the opposite change.

See *DB2 Administration Guide* for considerations for using SMS to manage data sets.

Examples

Example 1: Alter storage group DSN8G110. Add volumes DSNV04 and DSNV05.

```
ALTER STOGROUP DSN8G110
    ADD VOLUMES (DSNV04,DSNV05);
```

Example 2: Alter storage group DSN8G110. Remove volumes DSNV04 and DSNV05.

```
ALTER STOGROUP DSN8G110
  REMOVE VOLUMES (DSNV04,DSNV05);
```

ALTER TABLE

The ALTER TABLE statement changes the description of a table at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set that is defined below must include at least one of the following:
- The ALTER privilege on the table
- Ownership of the table
- DBADM authority for the database
- SYSADM or SYSCTRL authority
- System DBADM

To alter a system-period temporal table when one or more of the changes also result in changes to the associated history table, the privileges that are held by the authorization ID of the statement must also include at least one of the following:
- The ALTER privilege on the history table
- Ownership of the history table
- DBADM authority for the database
- SYSADM or SYSCTRL authority
- System DBADM

If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

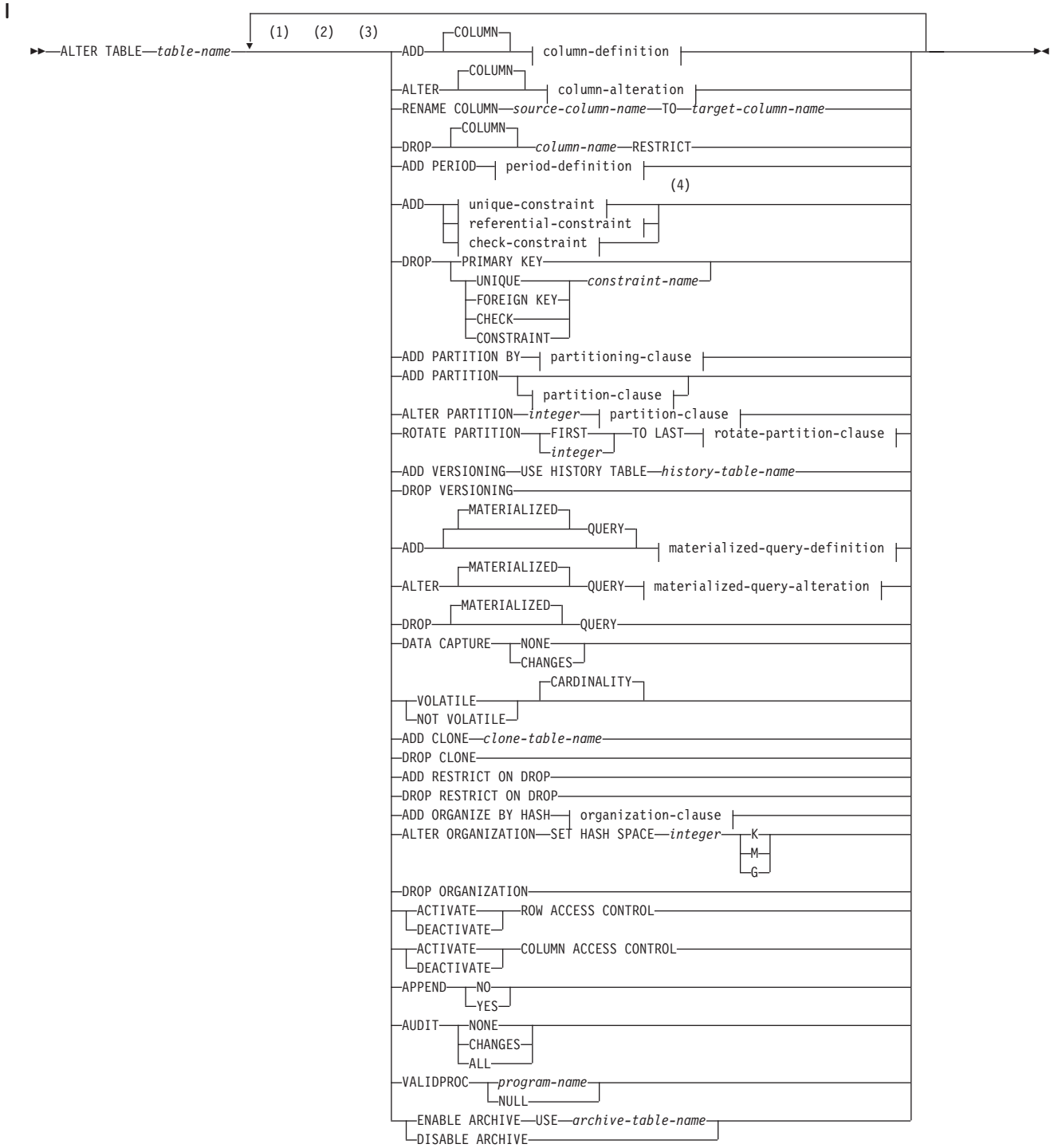
The privilege set must include SECADM authority if one of the following clauses is specified:
- ACTIVATE
- DEACTIVATE

Additional privileges might be required in the following situations:
- FOREIGN KEY, ADD PRIMARY KEY, ADD UNIQUE, DROP PRIMARY KEY, DROP FOREIGN KEY, or DROP CONSTRAINT is specified.
- The data type of a column that is added to the table is a distinct type.
- A fullselect is specified.
- A column is defined as a security label column.
- A column is defined as ROWID GENERATED BY DEFAULT.

See the description of the appropriate clauses for the details about these privileges.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the package. If the statement is dynamically prepared, the privilege set is the union of the privilege sets that are held by each authorization ID and role of the process.

Syntax



Notes:

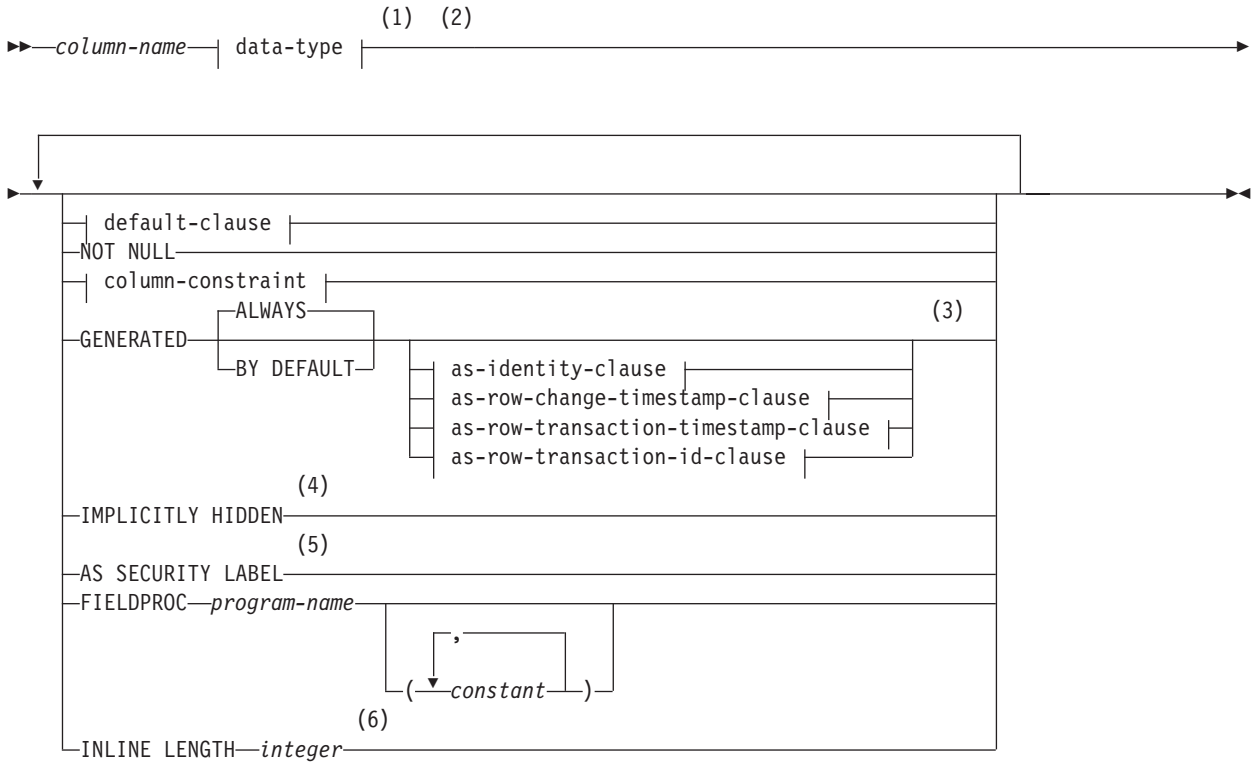
1 The same clause must not be specified more than one time, except for the ADD COLUMN or ALTER COLUMN clauses. If multiple ADD COLUMN clauses are specified in the same statement, at most one ADD COLUMN clause can contain a *references-clause*. If ALTER COLUMN SET DATA TYPE is specified, it must be specified first.

2 The ALTER COLUMN, ADD PARTITION, ALTER PARTITION, and ROTATE PARTITION clauses are mutually exclusive with each other.

3 If ADD CLONE, DROP CLONE, RENAME COLUMN, ADD ORGANIZE BY HASH, ALTER ORGANIZATION, DROP ORGANIZATION, ADD VERSIONING, DROP VERSIONING, DROP COLUMN, ACTIVATE, DEACTIVATE, ENABLE ARCHIVE, or DISABLE ARCHIVE is specified, no other clause is allowed on the ALTER TABLE statement.

4 The ADD keyword is optional for *referential-constraint* or *unique-constraint* if it is the first clause specified in the statement. Otherwise, ADD is required.

column-definition:



Notes:

1 *data-type* is optional if *as-row-change-timestamp-clause* is specified

2 The same clause must not be specified more than one time.

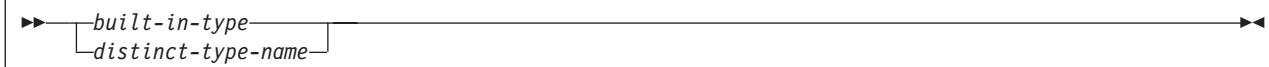
3 GENERATED must be specified if the column is to be an identity column.

4 IMPLICITLY HIDDEN must not be specified for a column defined as a ROWID, or a distinct type that is based on a ROWID.

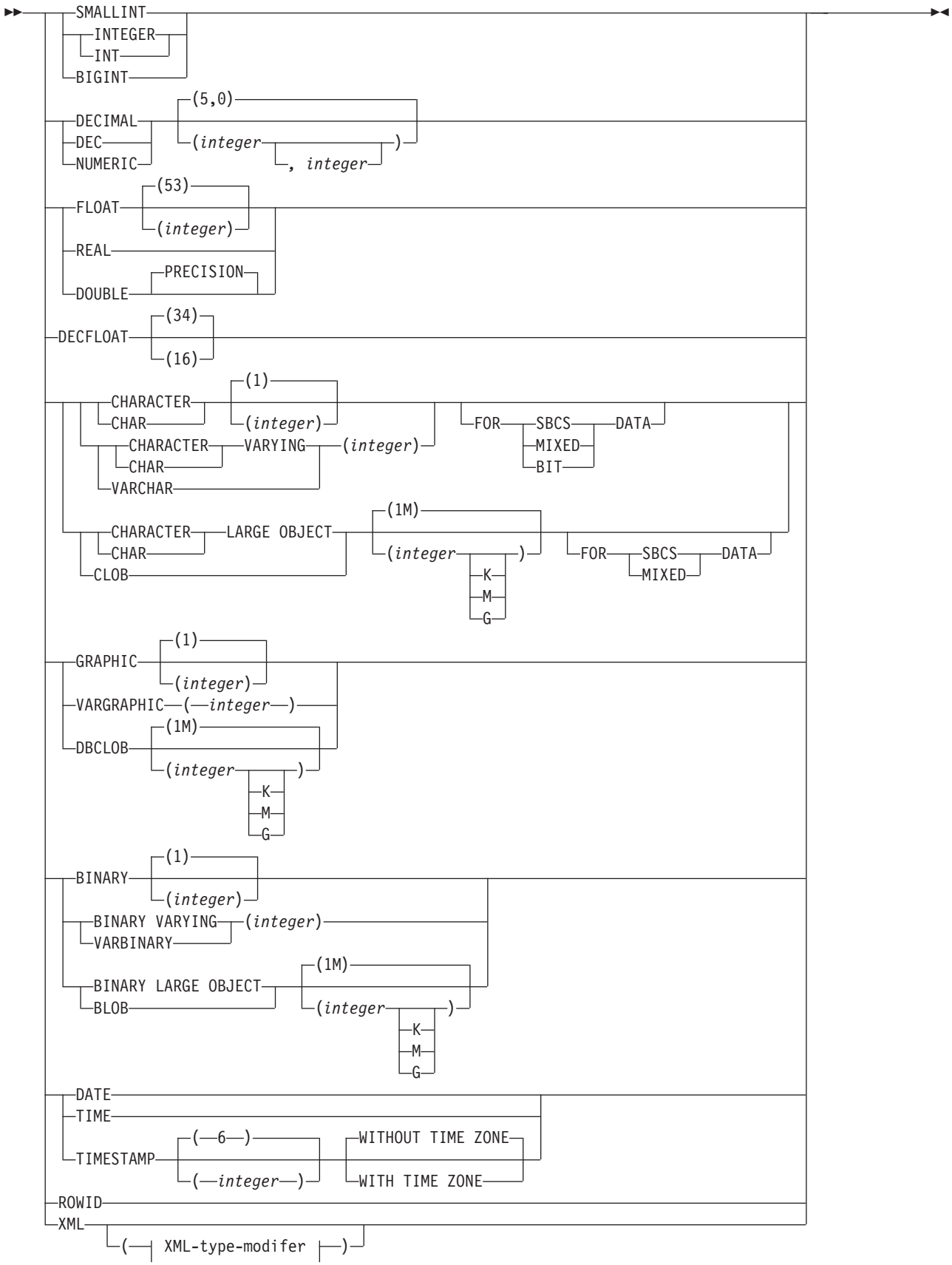
5 AS SECURITY LABEL can be specified only for a CHAR(8) data type and requires that the NOT NULL and WITH DEFAULT clauses be specified.

6 INLINE LENGTH only applies to a column with a LOB data type or a distinct type that is based on a LOB data type.

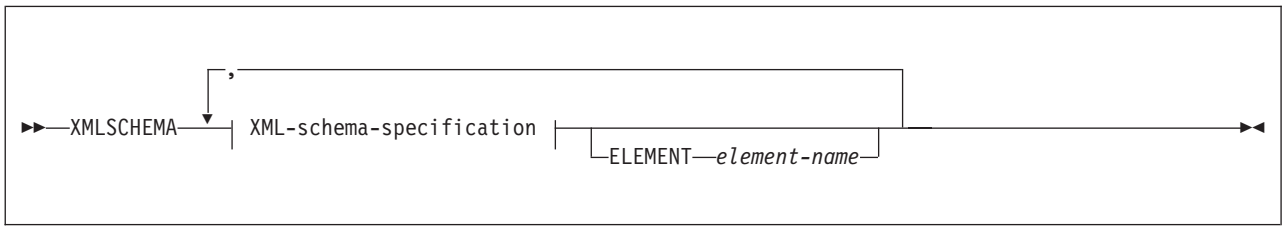
data-type:



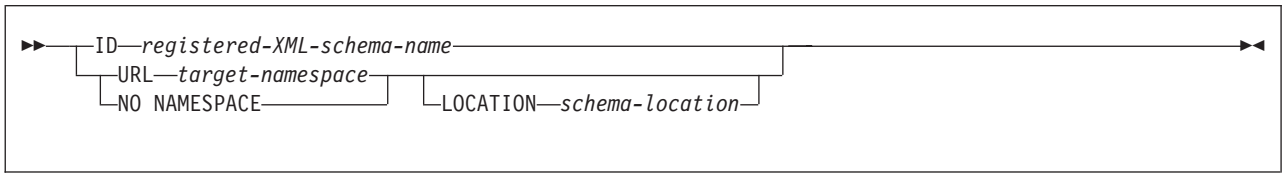
built-in-type:


SMALLINT
INTEGER
INT
BIGINT
DECIMAL
DEC
NUMERIC
(5,0)
(integer
, integer)
FLOAT
(53)
(integer)
REAL
DOUBLE
PRECISION
DECFLOAT
(34)
(16)
CHARACTER
CHAR
(1)
(integer)
CHARACTER
CHAR
VARYING
VARCHAR
(integer)
FOR
SBCS
MIXED
BIT
DATA
CHARACTER
CHAR
CLOB
LARGE OBJECT
(1M)
(integer
K
M
G)
FOR
SBCS
MIXED
DATA
GRAPHIC
(1)
(integer)
VARGRAPHIC
(integer)
DBCLOB
(1M)
(integer
K
M
G)
BINARY
(1)
(integer)
BINARY VARYING
VARBINARY
(integer)
BINARY LARGE OBJECT
BLOB
(1M)
(integer
K
M
G)
DATE
TIME
TIMESTAMP
(6)
(integer)
WITHOUT TIME ZONE
WITH TIME ZONE
ROWID
XML
(XML-type-modifer)

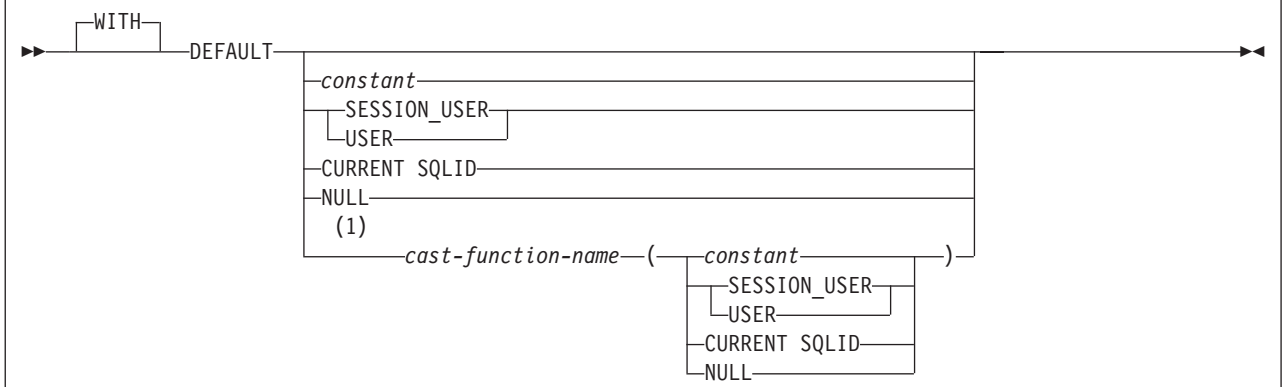
XML-type-modifer:



XML-schema-specification:



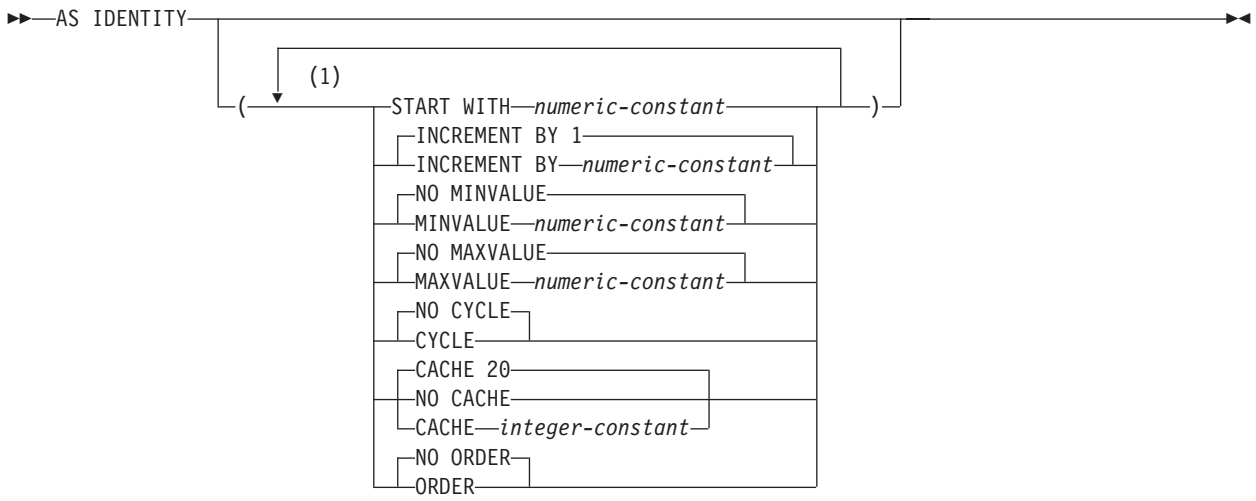
default-clause:



Notes:

1 The *cast-function-name* form of the DEFAULT value can only be used with a column that is defined as a distinct type.

as-identity-clause:



Notes:

1 Separator commas can be specified between attributes when an identity column is defined.

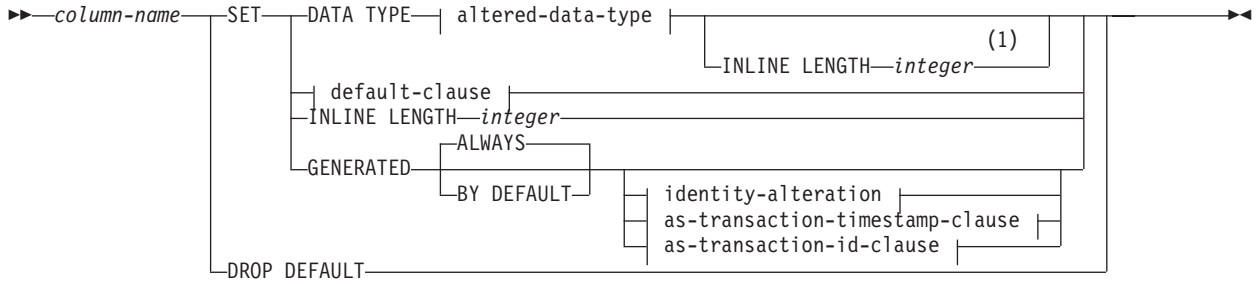
as-row-change-timestamp-clause:

as-row-transaction-timestamp-clause:

as-row-transaction-id-clause:

column-constraint

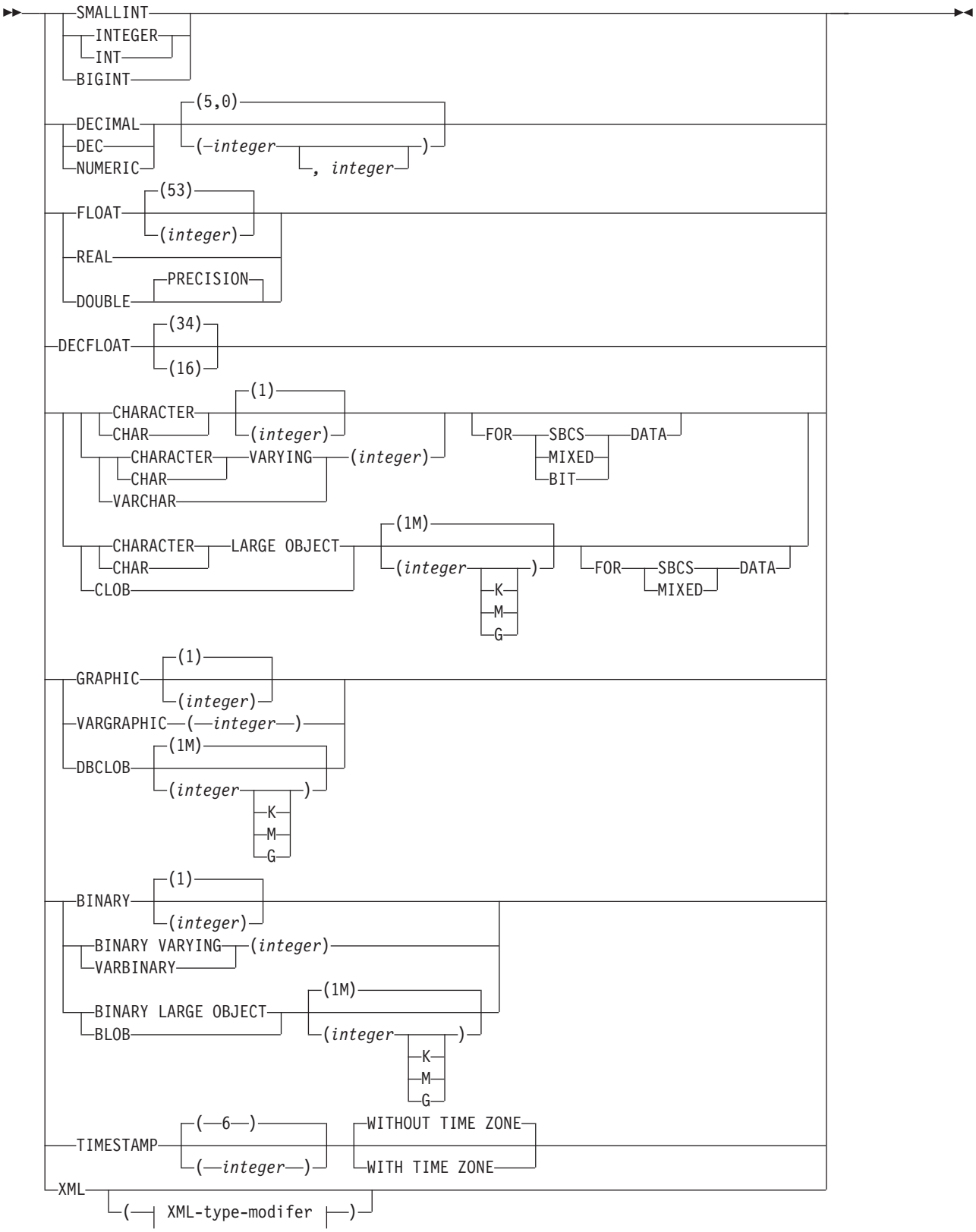
column-alteration:



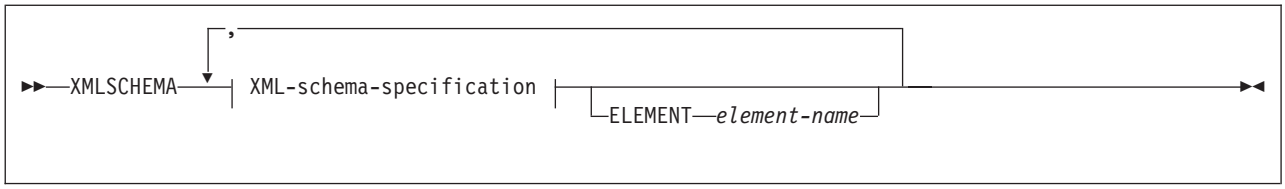
Notes:

1 INLINE LENGTH can only be specified for LOB columns in tables that are in universal table spaces. INLINE LENGTH cannot be specified if FOR SBCS DATA or FOR MIXED DATA is also specified.

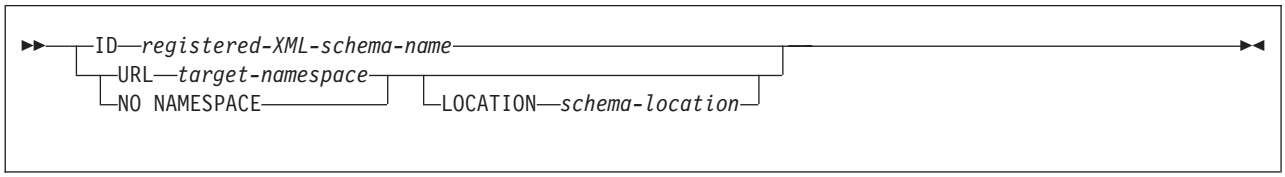
altered-data-type:



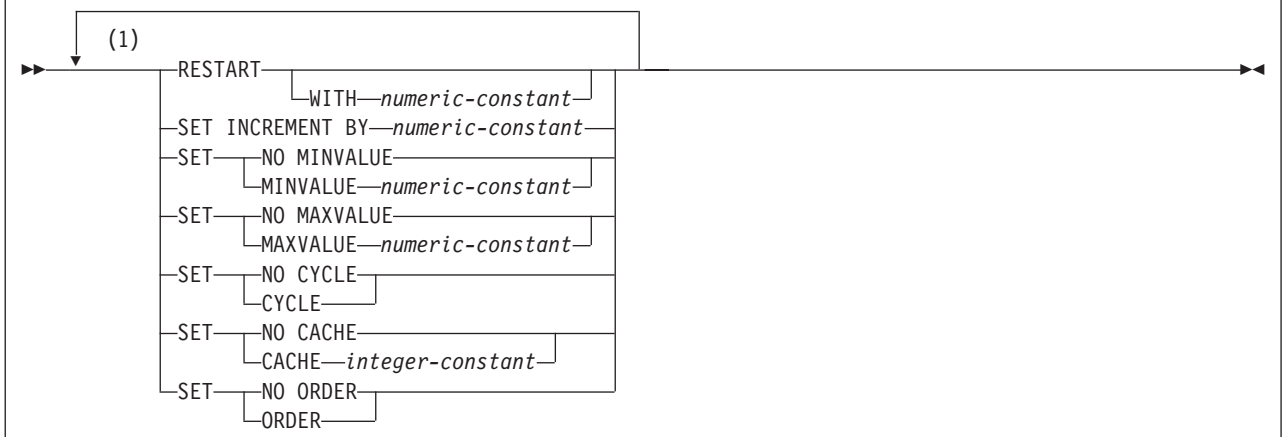
XML-type-modifer:



XML-schema-specification:



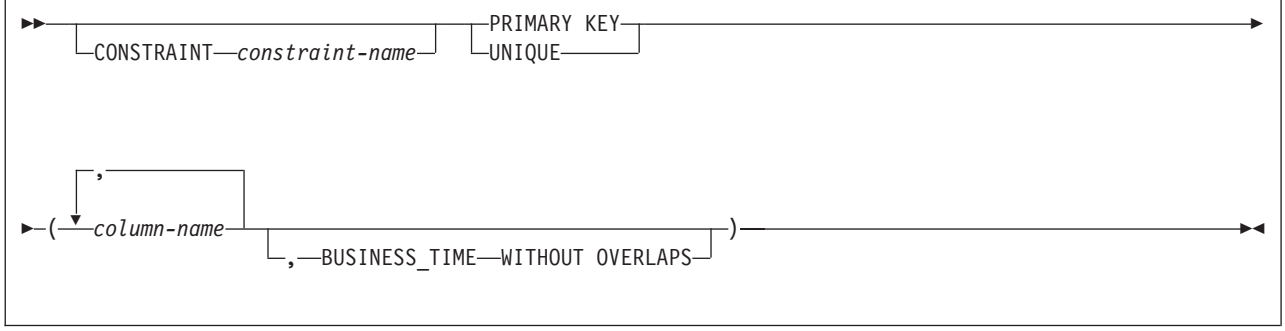
identity-alteration:



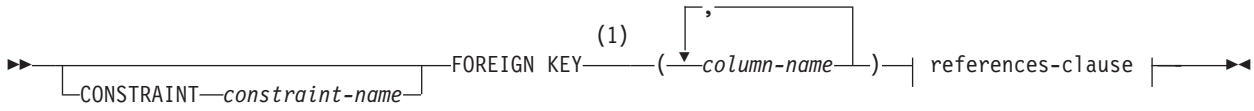
Notes:

1 At least one option must be specified and the same clause must not be specified more than one time.

unique-constraint:



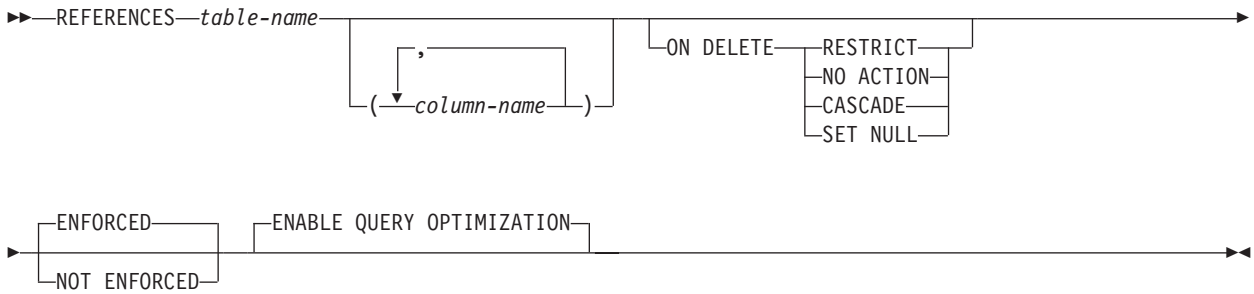
referential-constraint:



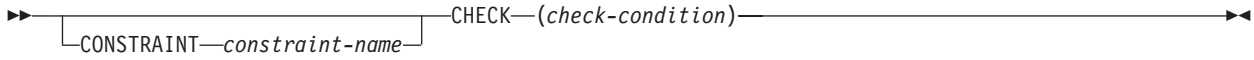
Notes:

1 For compatibility with prior releases, when the CONSTRAINT clause (shown above) is not specified, a *constraint-name* can be specified following FOREIGN KEY.

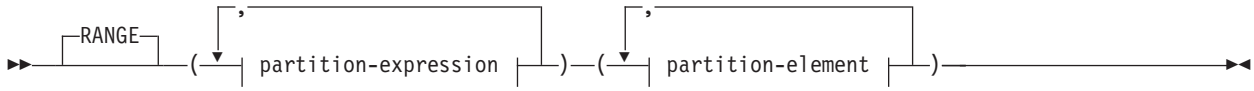
references-clause:



check-constraint:

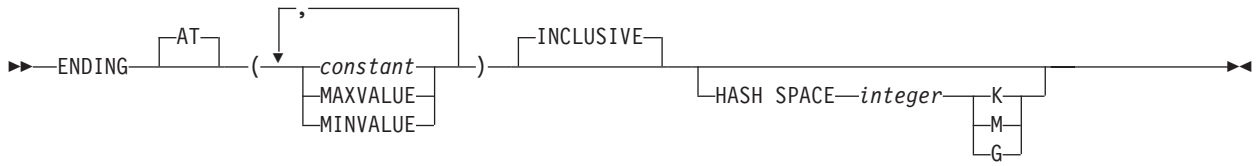


partitioning-clause:

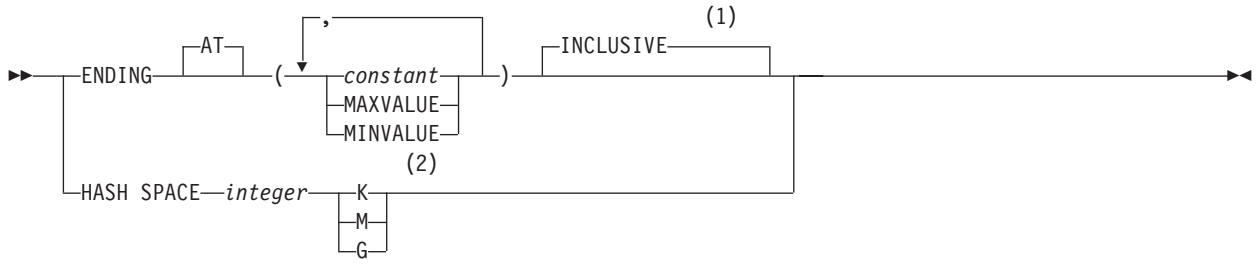


partition-expression:

partition-element:



partition-clause:



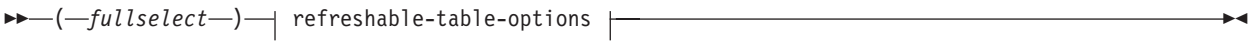
Notes:

1 The ENDING clause must not be specified for a partition-by-growth table space, but must be specified for a range partitioned table space.

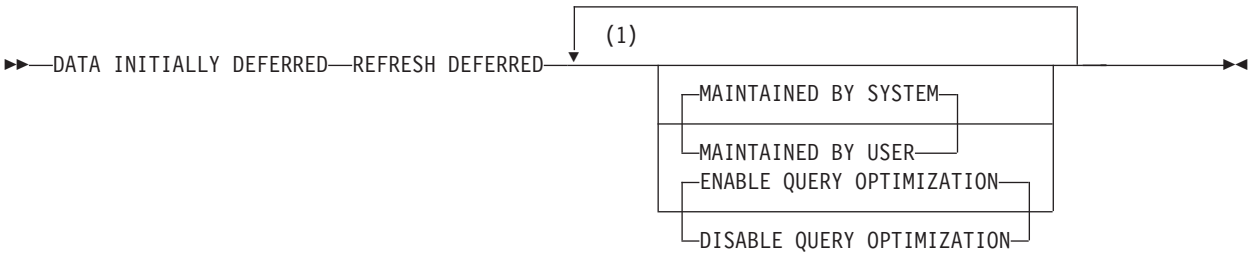
2 The HASH SPACE clause can only be specified for the ALTER PARTITION clause.

partition-rotation:

materialized-query-definition:



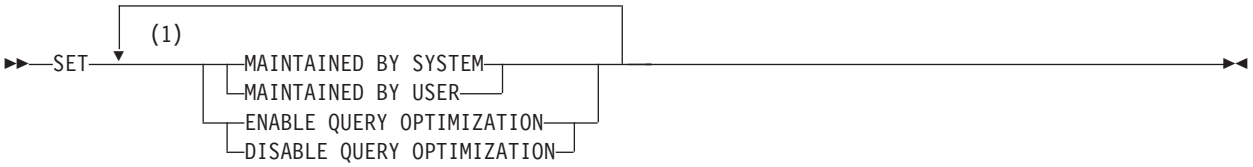
refreshable-table-options:



Notes:

1 The same clause must not be specified more than one time.

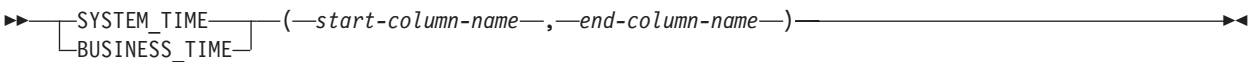
materialized-query-table-alteration:



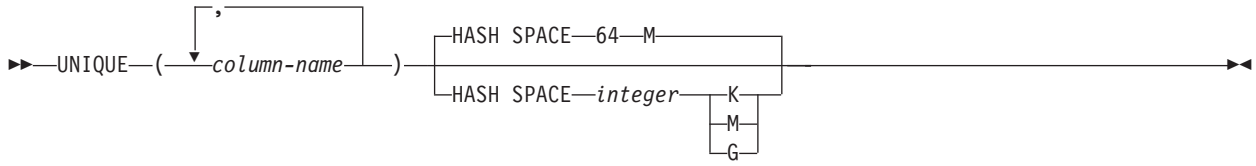
Notes:

1 The same clause must not be specified more than one time.

period-definition:



organization-clause:



Description

table-name

Identifies the table to be altered. The name must identify a table that exists at the current server. The name must not identify a declared temporary table, view, or a table that was implicitly created for an XML column. If the name identifies a catalog table, DATA CAPTURE CHANGES is the only clause that can be specified.

If *table-name* identifies an auxiliary table, alterations are limited to the following clauses:

- APPEND

If *table-name* identifies a materialized query table, alterations are limited to the following clauses:

- AUDIT
- DATA CAPTURE
- ALTER MATERIALIZED QUERY
- DROP MATERIALIZED QUERY
- ADD RESTRICT ON DROP
- DROP RESTRICT ON DROP

ADD COLUMN:

ADD *column-definition*

Adds a column to the table. Except for a ROWID column and an identity column, all values of the column in existing rows are set to its default value. If the table has *n* columns, the ordinality of the new column is *n*+1. The value of *n* cannot be greater than 749. For a dependent table, *n* cannot be greater than 748.

The column cannot be added if the increase in the total byte count of the columns exceeds the maximum row size. The maximum row size for the table is eight less than the maximum record size as described in Maximum record size.

If you add a LOB column and the table does not already have a ROWID column, DB2 creates an implicitly hidden ROWID column. For details about adding a LOB column, such as the other objects that might be implicitly created or need to be explicitly created, see Creating a table with LOB columns. For more information about adding a ROWID column, see Adding a ROWID column.

For implicitly created LOB objects, the privilege set requires CREATETAB and CREATETS privileges on the database that contains the table (DSNDB04 if the database is implicitly created) and the USE privilege on the buffer pool and the

storage group that is used by the auxiliary table and the LOB table space. The implicitly created objects are owned by the owner of the base table.

If you add an XML column, the privilege set requires the CREATETAB and CREATETS privileges on the database that contains the table (DSNDB04 if the database is implicitly created), INDEX on the base table for the first DOCID column that is added, and USE privilege on the buffer pool and the storage group that is used by the XML objects. These privileges are required for implicitly created XML objects. The implicitly created objects are owned by the owner of the base table.

When you add a column to a table, the table space is placed into advisory REORG-pending status.

The table must not be a history table or archive table.

If the table is a system-period temporal table, the column is also added to the associated history table. If the table is an archive-enabled table, the column is also added to the associated archive table. The following attributes of the column in the associated table are the same as the attributes of the corresponding column of the table that is being altered:

- Name
- Data type
- Length (including inline LOB lengths), precision, scale
- FOR BIT, SBCS, or MIXED DATA attribute for a character string column
- Null attribute
- Hidden attribute
- Field procedure

You cannot add the following columns:

- A column to a table that has an edit procedure that is defined as WITH ROW ATTRIBUTES.
- A ROWID column to a table that already has an explicitly defined ROWID column
- An identity column to a table that has an identity column
- A security label column to a table that already has a security label column
- A security label column to a system-period temporal table or archive-enabled table
- A row change timestamp column to a table that already has a row change timestamp column
- A LOB, ROWID, identity column, or row change timestamp column to a created temporary table
- A GRAPHIC, VARGRAPHIC, DBCLOB, or CHAR FOR MIXED DATA column, when the setting for installation option MIXED DATA is NO
- A Unicode column to an EBCDIC table (specifying CCSID 1208 or CCSID 1200) if the table is already defined with an EDITPROC or VALIDPROC.

If the column that is being added is a security label column, row permissions, including the default row permission, cannot exist for the table

column-name
Names of the column you want to add to the table. The name must not be the same as the name of an existing column of the table or the name of a period in the table. A column named SYSTEM_TIME or BUSINESS_TIME cannot be added to a table that is defined as a system-period temporal table or a history table. Do not qualify *column-name*.

built-in-type
Specifies the data type of the column is one of the built-in data types. See built-in-type for information about the built-in data types that can be used when adding a column to a table.

distinct-type-name
Specifies the distinct type (user-defined data type) of the column. The length and scale of the column are respectively the length and scale of the source type of the distinct type. The privilege set must implicitly or explicitly include the USAGE privilege on the distinct type.

The encoding scheme of the distinct type must be the same as the encoding scheme of the table.

If the column is to be used in the definition of the foreign key of a referential constraint, the data type of the corresponding column of the parent key must have the same distinct type.

DEFAULT
Specifies the default value that is assigned to the column in the absence of a value specified in a data change statement, or LOAD. Do not specify DEFAULT for the following types of columns:
- A ROWID column (DB2 generates default values)
- An identity column (DB2 generates default values)
- An XML column
- A row change timestamp column

Do not specify a value after the DEFAULT keyword for a security label column. DB2 provides the default for a security label column.

If a CCSID clause is specified for the column, do not specify a value after the DEFAULT keyword. Alternatively, DEFAULT NULL can be specified.

If a value is not specified after the DEFAULT keyword, the default value depends on the data type of the column as indicated in the following table:

Data Type
Default Value

Numeric
0

Fixed-length character or graphic string
Blanks

Fixed-length binary string
Hexadecimal zeros

Varying-length string
A string of length 0

Inline BLOB
Hexadecimal zeros

Inline CLOB
Blanks

Inline DBCLOB
Blanks

Date For existing rows, a date corresponding to 1 January 0001. For added rows, CURRENT DATE.

Time For existing rows, a time corresponding to 0 hours, 0 minutes, and 0 seconds. For added rows, CURRENT TIME.

Timestamp without time zone
For existing rows, a date corresponding to 1 January 0001, and a time corresponding to 0 hours, 0 minutes, 0 seconds, and zeros for fractional seconds up to the timestamp precision. For added rows, CURRENT_TIMESTAMP(*p*) WITHOUT TIME ZONE where *p* is the corresponding timestamp precision.

Timestamp with time zone
For existing rows, a date corresponding to 1 January 0001, and a time corresponding to 0 hours, 0 minutes, 0 seconds, and zeros for fractional seconds up to the timestamp precision, 0 time zone hours, 0 time zone minutes. For added rows, CURRENT_TIMESTAMP(*p*) WITH TIME ZONE where *p* is the corresponding timestamp precision.

If the column is defined as timestamp with time zone, the default value must include a time zone.

In a given column definition:

- DEFAULT and FIELDPROC cannot both be specified.
- NOT NULL and DEFAULT NULL cannot both be specified.
- Omission of NOT NULL and DEFAULT for a column other than an identity column is an implicit specification of DEFAULT NULL. For an identity column, it is an implicit specification of NOT NULL, and DB2 generates default values.

A default value other than the one that is listed above can be specified in one of the following forms:

- WITH DEFAULT for a default value of an empty string
- DEFAULT NULL for a default value of null

constant
Specifies a constant as the default value for the column. The value of the constant must conform to the rules for assigning that value to the column.

A character or string constant must be short enough so that its UTF-8 representation requires no more than 1536 bytes. A hexadecimal graphic string (GX) constant cannot be specified.

In addition, the length of the constant value cannot be greater than the INLINE LENGTH attribute for LOB columns.

SESSION_USER or USER
Specifies the value of the SESSION_USER (USER) special register at the time of an SQL data change statement or LOAD, as the default for the column. If SESSION_USER is specified, the data type of the column must be a character string with a length attribute greater than or equal to 8 characters when the value is expressed in CCSID 37. If the data type of the column is an inline CLOB, the INLINE LENGTH attribute must be greater than or equal to 8 characters when the value is expressed as CCSID 37. For existing rows, the value is that of the SESSION_USER special register at the time the ALTER TABLE statement is processed.

CURRENT SQLID
Specifies the value of the SQL authorization ID of the process at the

time of an SQL data change statement or LOAD, as the default for the column. If CURRENT SQLID is specified, the data type of the column must be a character string with a length attribute greater than or equal to the length attribute of the CURRENT SQLID special register. If the data type of the column is an inline CLOB, the INLINE LENGTH attribute must be greater than or equal to the length attribute of the CURRENT SQLID special register. For existing rows, the value is the SQL authorization ID of the process at the time the ALTER TABLE statement is processed.

NULL
Specifies null as the default value for the column.

cast-function-name
The name of the cast function that matches the name of the distinct type for the column. A cast function can be specified only if the data type of the column is a distinct type.

The schema name of the cast function, whether it is explicitly specified or implicitly resolved through function resolution, must be the same as the explicitly or implicitly specified schema name of the distinct type.

constant
Specifies a constant as the argument. The constant must conform to the rules of a constant for the source type of the distinct type. The length of the constant cannot be greater than the INLINE LENGTH attribute for LOB columns.

SESSION_USER or USER
Specifies the value of the SESSION_USER (USER) special register at the time a row is inserted as the default for the column. The source type of the distinct type of the column must be a CHAR, VARCHAR, or inline CLOB with a length attribute (inline length attribute for CLOB) that is greater than or equal to the length attribute of the SESSION_USER special register.

CURRENT SQLID
Specifies the value of the CURRENT SQLID special register at the time a row is inserted as the default for the column. The source type of the distinct type of the column must be a CHAR, VARCHAR, or inline CLOB with a length attribute (or inline length attribute for CLOB) that is greater than or equal to the length attribute of the CURRENT SQLID special register.

NULL
Specifies the NULL value as the argument.

GENERATED
Specifies that DB2 generates values for the column. GENERATED is applicable only to ROWID columns, identity columns, row change timestamp columns, *row-begin* columns, *row-end* columns, and *transaction-start-ID* columns. If the table is a system-period temporal table, GENERATED must not be specified for the column that is to be added, unless the column is a ROWID column. The default is GENERATED ALWAYS.

ALWAYS
Specifies that DB2 will generate a value for the column when a row is inserted into the table. ALWAYS is the recommended value unless you are using data propagation.

BY DEFAULT
Specifies that DB2 will generate a value for the column when a row is inserted unless a value was specified for the column on the data change statement.

If a user-supplied value is specified for a ROWID column, DB2 uses the value only if it is a valid row ID value that was previously generated by DB2 and the column has a unique, single-column index. Until this index is created on the ROWID column, the insert, and update operations and the LOAD utility cannot be used to add rows to the table. If the table space name is not specified on the CREATE TABLE statement, DB2 implicitly creates the necessary object to make the table complete, including the index. The name of this index is 'I' followed by the first ten characters of the column name followed by seven randomly generated characters. If the column name is less than ten characters, DB2 adds underscore characters to the end of the name until it has ten characters. The implicitly created index has the COPY NO attribute.

For an identity column, DB2 inserts a specified value but does not verify that it is a unique value for the column unless the identity column has a unique, single-column index.

If a user-supplied value is specified for an identity column, DB2 inserts the specified value but does not perform any special validation on that value beyond the normal validation that is performed for any column. DB2 does not check how the specified value affects the sequential properties that are defined for the identity column. To ensure the uniqueness of an identity column that is defined as GENERATED BY DEFAULT, define a unique index on the identity column.

BY DEFAULT is the recommended value only when you are using data propagation.

AS IDENTITY
Specifies that the column is an identity column for the table. A table can have only one identity column. AS IDENTITY can be specified only if the data type for the column is an exact numeric type with a scale of zero (SMALLINT, INTEGER, BIGINT, DECIMAL with a scale of zero, or a distinct type based on one of these types). Separator commas between identity column attribute specifications are optional when the identity column is defined.

An identity column is implicitly NOT NULL. When adding an identity column to a table, you must also specify GENERATED ALWAYS or GENERATED BY DEFAULT.

Defining a column AS IDENTITY does not necessarily guarantee uniqueness of the values. To ensure uniqueness of the values, define a unique, single-column index on the identity column.

START WITH *numeric-constant*
Specifies the first value that is generated for the identity column. The value can be any positive or negative value that could be assigned to the column without non-zero digits existing to the right of the decimal point.

If a value is not explicitly specified when the identity column is defined, the default is the MINVALUE for an ascending identity column and the MAXVALUE for a descending identity column.

This value is not necessarily the value that would be cycled to after reaching the maximum or minimum value for the identity column. The START WITH clause can be used to start the generation of values outside the range that is used for cycles. The range used for cycles is defined by MINVALUE and MAXVALUE.

INCREMENT BY *numeric-constant*
Specifies the interval between consecutive values of the identity column. The value can be any positive or negative value (including 0) that does not exceed the value of a large integer constant and could be assigned to the column without any non-zero digits existing to the right of the decimal point. The default is 1.

If the value is positive or zero, the sequence of values for the identity column ascends. If the value is negative, the sequence of values descends.

MINVALUE or NO MINVALUE
Specifies the minimum value at which a descending identity column either cycles or stops generating values or an ascending identity column cycles to after reaching the maximum value.

NO MINVALUE
Specifies that the minimum end point of the range of values for the identity column has not be set. In such a case, the default value for MINVALUE becomes one of the following:

- For an ascending identity column, the value is the START WITH value or 1 if START WITH was not specified.
- For a descending identity column, the value is the minimum value of the data type of the column.

MINVALUE *numeric-constant*
Specifies the numeric constant that is the minimum value that is generated for this identity column. This value can be any positive or negative value that could be assigned to this column without non-zero digits existing to the right of the decimal point. The value must be less than or equal to the maximum value.

MAXVALUE or NO MAXVALUE
Specifies the maximum value at which a ascending identity column either cycles or stops generating values or a descending identity column cycles to after reaching the minimum value.

NO MAXVALUE
Specifies that the minimum end point of the range of values for the identity column has not be set. In such a case, the default value for MAXVALUE becomes one of the following:

- For an ascending identity column, the value is the maximum value of the data type of the column.
- For a descending identity column, the value is the START WITH value or -1 if START WITH is not specified.

MAXVALUE *numeric-constant*
Specifies the numeric constant that is the maximum value that is generated for this identity column. This value can be any positive or negative value that could be assigned to this

column without non-zero digits existing to the right of the decimal point. The value must be greater than or equal to the minimum value.

CYCLE or NO CYCLE
Specifies whether this identity column should continue to generate values after reaching either its maximum or minimum value.

NO CYCLE
Specifies that values will not be generated for the identity column after the maximum or minimum value has been reached. This is the default.

CYCLE
Specifies that values continue to be generated for this column after the maximum or minimum value has been reached. If this option is used, after an ascending identity column reaches the maximum value, it generates its minimum value. After a descending identity column reaches its minimum value, it generates its maximum value. The maximum and minimum values for the identity column determine the range that is used for cycling.

When CYCLE is in effect, duplicate values can be generated by DB2 for an identity column. However, if a unique index exists on the identity column and a non-unique value is generated for it, an error occurs.

CACHE or NO CACHE
Specifies whether to keep some preallocated values in memory. Preallocating and storing values in the cache improves the performance of inserting rows into a table. The default is CACHE 20.

NO CACHE
Specifies that values for the identity column are not preallocated and stored in the cache, ensuring that values will not be lost in the case of a system failure. In this case, every request for a new value for the identity column results in synchronous I/O.

CACHE *integer-constant*
Specifies the maximum number of values of the identity column sequence that DB2 can preallocate and keep in memory.

During a system failure, all cached identity column values that are yet to be assigned might be lost and will not be used. Therefore, the value that is specified for CACHE also represents the maximum number of values for the identity column that could be lost during a system failure.

The minimum value is 2.

In a data sharing environment, you can use the CACHE and NO ORDER options to allow multiple DB2 members to cache sequence values simultaneously.

ORDER or NO ORDER
Specifies whether the identity column values must be generated in order of request. The default is NO ORDER.

NO ORDER
Specifies that the values do not need to be generated in order of request.

ORDER
Specifies that the values are generated in order of request. Specifying ORDER might disable the caching of values. ORDER applies only to a single-application process.

In a data sharing environment, if the CACHE and NO ORDER options are in effect, multiple caches can be active simultaneously, and the requests for identity values from different DB2 members might not result in the assignment of values in strict numeric order. For example, if members DB2A and DB2B are using the identity column, and DB2A gets the cache values 1 to 20 and DB2B gets the cache values 21 to 40, the actual order of values assigned would be 1,21,2 if DB2A requested a value first, then DB2B requested, and then DB2A again requested. Therefore, to guarantee that identity values are generated in strict numeric order among multiple DB2 members using the same identity column, specify the ORDER option.

FOR EACH ROW ON UPDATE AS ROW CHANGE TIMESTAMP
Specifies that the column is a timestamp and the values will be generated by DB2. DB2 generates a value for the column for each row as a row is inserted, and for any row for which any column is updated. The value that is generated for a row change timestamp column is a timestamp that corresponds to the time of the insert or update of the row. If multiple rows are inserted or updated with a single statement, the value of the row change timestamp column might be different for each row.

If *data-type* is specified, it must be TIMESTAMP WITHOUT TIME ZONE with a precision of 6. You must specify NOT NULL with a row change timestamp column.

AS ROW BEGIN
Specifies that a value for the data type of the column is assigned when a row is inserted or any column in the row is updated. The value that is assigned for a TIMESTAMP WITHOUT TIME ZONE column is TIMESTAMP value '9999-12-30-00.00.00.000000000000'. The value that is assigned for a TIMESTAMP WITH TIME ZONE COLUMN is TIMESTAMP value '9999-12-30.00.00.00.000000000000 +00:00'.

A *row-begin* column is intended to be used for a system-period temporal table.

A table can have only one *row-begin* column. If *data-type* is not specified, the column is defined as TIMESTAMP(12) WITHOUT TIME ZONE. If *data-type* is specified, it must be TIMESTAMP(12) WITHOUT TIME ZONE or TIMESTAMP(12) WITH TIME ZONE. If the column is defined as TIMESTAMP WITH TIME ZONE, the values are stored in UTC, with a time zone of +00:00. The column cannot have a DEFAULT clause.

A *row-begin* column is not updatable.

AS ROW END
Specifies that a value for the data type of the column is assigned when a row is inserted or any column in the row is updated. The value that is assigned for a timestamp without time zone column is TIMESTAMP

'9999-12-30-00.00.00.000000000000'. The value that is assigned for a timestamp with time zone column is TIMESTAMP '9999-12-30.00.00.00.000000000000 +00:00'.

A *row-end* column is intended to be used for a system-period temporal table.

For a table with system-period data versioning, when a row is deleted as the result of an update or delete operation, the value of the *row-end* column in the historical row reflects when the row was deleted. The value that is generated for the column in the historical row is a timestamp that corresponds to the most recent transaction start time that is associated with the transaction. If a row that is to be updated would result in a value for the *row-end* column that is less than or equal to the value for the corresponding *row-begin* column, the timestamp value for the *row-end* column is adjusted. If multiple rows are deleted with a single SQL statement, the values for the column in the historical rows are the same.

A table can have only one *row-end* column. If *data-type* is not specified, the column is defined as TIMESTAMP(12) WITHOUT TIME ZONE. If *data-type* is specified, it must be TIMESTAMP(12) WITHOUT TIME ZONE or TIMESTAMP(12) WITH TIME ZONE. If the column is defined as TIMESTAMP WITH TIME ZONE, the values are stored in UTC, with a time zone of +00:00. The column cannot have a DEFAULT clause.

A *row-end* column is not updatable.

AS TRANSACTION START ID
Specifies that a timestamp value is assigned when the row is inserted or any column in the row is updated. If the value of the *row-begin* column is unique from *row-begin* column values that are generated for other transactions, the *row-begin* column value is assigned to the *transaction-start-ID* column. Otherwise, the value of the *transaction-start-ID* column is derived from the *row-begin* column value and adjusted to make it unique from *transaction-start-ID* column values that are generated for other transactions.

A *transaction-start-ID* column is intended to be used for a system-period temporal table.

A table can have only one *transaction-start-id* column. If *data-type* is not specified, the column is defined as TIMESTAMP(12) WITHOUT TIME ZONE. If *data-type* is specified, it must be TIMESTAMP(12) WITHOUT TIME ZONE or TIMESTAMP(12) WITH TIME ZONE. If the column is defined as TIMESTAMP WITH TIME ZONE, the values are stored in UTC, with a time zone of +00:00. The column cannot have a DEFAULT clause.

A *transaction-start-id* column is not updatable.

NOT NULL
Prevents the column from containing null values. If NOT NULL is specified, the DEFAULT clause must be used to specify a nonnull default value for the column unless the column has a row ID data type or is an identity column. For a ROWID column, NOT NULL must be specified, and DEFAULT must not be specified. For an identity column, although NOT NULL can be specified, DEFAULT must not be specified.

IMPLICITLY HIDDEN
Specifies that the column is not visible in the results of SQL statements unless you refer explicitly to the column by name. For example, assume that table T1 includes a column that is defined with the IMPLICITLY HIDDEN clause. The result of `SELECT * FROM T1` would not include the implicitly hidden column. However, the result of a SELECT statement that explicitly refers to the name of the implicitly hidden column would include that column in the result table.

IMPLICITLY HIDDEN must not be specified for a column that is defined as a ROWID, or a distinct type that is based on a ROWID.

references-clause
The *references-clause* of a *column-definition* provides a shorthand method of defining a foreign key composed of a single column. Thus, if *references-clause* is specified in the definition of column C, the effect is the same as if that *references-clause* were specified as part of a FOREIGN KEY clause in which C is the only identified column.

Do not specify *references-clause* in the definition of the following types of columns because these types of columns cannot be a foreign key:

- LOB columns
- ROWID columns
- XML columns
- DECFLOAT columns
- Row change timestamp columns
- Security label columns
- Columns defined with a CCSID clause

check-constraint
The *check-constraint* of a *column-definition* has the same effect as specifying a check constraint in a separate ADD *check-constraint* clause. For conformance with the SQL standard, a check constraint specified in the definition of column C should not reference any columns other than C.

Do not specify a check constraint in the definition of the following types of columns:

- LOB columns
- ROWID columns
- XML columns
- DECFLOAT columns
- Security label columns
- Columns defined with a CCSID clause

FIELDPROC *program-name*
Designates *program-name* as the field procedure exit routine for the column. A field procedure can be specified only for a column with a length attribute that is not greater than 255 bytes. FIELDPROC can only be specified for columns that are a built-in character string or graphic string data types. The column must not be one of the following:

- a LOB column
- a security label column
- a row change timestamp column
- a column with the TIMESTAMP WITH TIME ZONE data type

- a Unicode column in an EBCDIC table

The field procedure encodes and decodes column values. Before a value is inserted in the column, it is passed to the field procedure for encoding. Before a value from the column is used by a program, it is passed to the field procedure for decoding. A field procedure could be used, for example, to alter the sorting sequence of values entered in the column.

The field procedure is also invoked during the processing of the ALTER TABLE statement. When so invoked, the procedure provides DB2 with the column's *field description*. The field description defines the data characteristics of the encoded values. By contrast, the information you supply for the column in the ALTER TABLE statement defines the data characteristics of the decoded values.

If you omit FIELDPROC, the column has no field procedure.

Related information:

Field procedures (DB2 Administration Guide)

constant

Is a parameter that is passed to the field procedure when it is invoked. A parameter list is optional. The *n*th parameter specified in the FIELDPROC clause on ALTER TABLE corresponds to the *n*th parameter of the specified field procedure. The maximum length of the parameter list is 255 bytes, including commas but excluding insignificant blanks and the delimiting parentheses.

AS SECURITY LABEL

Specifies that the table is defined with multilevel security with row level granularity and specifies that the column will contain the security label values. A table can have only one security label column. To define a table with a security label column, the primary authorization ID of the statement must have a valid security label, and the RACF SECLABEL class must be active. In addition, the following conditions are also required:

- The data type of the column must be CHAR(8).
- The subtype of the column must be SBCS.
- The column does not have any field procedures, check constraints, or referential constraints.
- The column must be defined as NOT NULL and WITH DEFAULT clauses.
- The WITH DEFAULT clause must not be specified with a default value (DB2 provides the default value).
- The table does not have an edit procedure that is defined as WITH ROW ATTRIBUTES.
- The table is not the source table for a materialized query table.

For existing rows in the table, the value of the security label column defaults to the security label of the user at the time the ALTER statement is executed.

INLINE LENGTH *integer*

Specifies the maximum length for the column, if the column is a LOB column and the table is in a universal table space. INLINE LENGTH cannot be specified if the column is not a LOB column (or a distinct type that is based on a LOB) or if the table is not in a universal table space.

For BLOB and CLOB columns, *integer* specifies the maximum number of bytes that are stored in the base table space for the column. *integer* must be between 0 and 32680 (inclusive) for a BLOB or CLOB column.

For a DBCLOB column, *integer* specifies the maximum number of double-byte characters that are stored in the table space for the column. *integer* must be between 0 and 16340 (inclusive) for a DBCLOB column.

If INLINE LENGTH is specified, the value of *integer* cannot be greater than the maximum length of the LOB column.

If the INLINE LENGTH clause is not specified, the maximum length of the LOB column depends on the following conditions:

- If a distinct type is not used or the distinct type that is used has been created without the INLINE LENGTH attribute, the LOB column will use the value of the LOB INLINE LENGTH parameter on installation panel DSNTIPD as the default inline length when the value of LOB INLINE LENGTH does not exceed the maximum length of the LOB column. If the value of LOB INLINE LENGTH exceeds the maximum length of the LOB column, the maximum length is the inline length of this LOB column.
- If a distinct type that has been created with the INLINE LENGTH attribute is used, the LOB column inherits the inline length from the distinct type.

Regardless of how the length is determined, the inline length of the LOB cannot be greater than its maximum length.

ALTER COLUMN:

ALTER COLUMN *column-alteration*

Alters the definition of an existing column, including the attributes of an existing identity column. Only the attributes specified are altered. Other attributes remain unchanged. Only future values of the column are affected by the changes made with an ALTER TABLE ALTER COLUMN statement.

The table being altered must not be in an incomplete state because of a missing unique index on a unique constraint (primary or unique key). An ALTER TABLE ALTER COLUMN statement might not be processed in the same unit of work as a data change statement. A column cannot be altered if any of the following conditions are true:

- The table has an edit procedure that is defined as WITH ROW ATTRIBUTES or a validation exit procedure
- The table is used in a materialized query table definition
- The table is a materialized query table
- The table is a system-period temporal table that is enabled for system-period data versioning
- The table is a history table
- The table is an archive-enabled table or an archive table
- There is an extended index that depends on that column
- The column is referenced in a field procedure
- The column is referenced in a referential constraint
- The column is referenced in the definition of a SYSTEM_TIME or BUSINESS_TIME period
- The column is defined as a *transaction-start-ID* column
- The column is defined as a security label column
- The column is defined as a row change timestamp column
- The column is a Unicode column in an EBCDIC table

You can modify all the attributes of an existing identity column, except for the data type of the column. To change the data type of an identity column, drop the table containing the column and recreate it. When the attributes of an identity column are altered, the column of the specified *column-name* must exist in the specified table and must have been defined with the IDENTITY attribute.

column-name
Identifies the column to be altered. The name must not be qualified and must identify an existing column in the table being altered when the ALTER statement is processed. The name must not identify a column that is being added in the same ALTER TABLE statement.

A column can only be referenced in one ALTER COLUMN clause in a single ALTER TABLE statement. However, that same column can be referenced multiple times for adding or dropping constraints in the same ALTER TABLE statement.

SET DATA TYPE *(altered-data-type)*
Specifies the new data type of the column to be altered. For a character column, you can also use the clause to change the definition of the subtype that is stored in the DB2 catalog and OBD.

The column cannot be an identity column. The new data type must be compatible with the existing data type of the column. The existing data type of the column cannot be a ROWID, date, time, or distinct type. When the source data type is a LOB, the target data type must be the same LOB data type. If the source data type is a LOB and the maximum length is altered, the new maximum length must be at least as large as the existing length attribute. If the column is a partitioning column, and the existing data type is CHAR or VARCHAR FOR BIT DATA, the new data type cannot be VARBINARY or BINARY. If the column is CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, or BINARY, the new data type cannot be VARBINARY if the column is part of an index and is defined with the DESC attribute. For more information on the compatibility of data types, see "Assignment and comparison" on page 121.

A TIMESTAMP column can only be altered to TIMESTAMP with a larger precision. A TIMESTAMP WITH TIME ZONE column can only be altered to TIMESTAMP WITH TIME ZONE with a larger precision. If the precision of a timestamp column is increased, the fractional seconds of existing data values are extended with zeros so that the number of fractional second digits matches the specified timestamp precision.

If the data type is a LOB and the maximum length is being changed, any packages or statements in the dynamic statement cache that reference the table are invalidated. Any views that reference the LOB column are regenerated.

If any numeric data type is being converted to DECFLOAT, the ALTER statement will fail if there is a partitioning key, check constraints, index, or a unique constraint on the column.

If *altered-data-type* is XML, the old data type of the altered column must also be XML:

- If the old data type has no XML type modifier and the new data type does, you should ensure that all values in the XML column are valid according to the XML schema that is specified in the type modifier. The XML table space for the column that is being changed is left in CHECK-pending status.

- If the old data type has the XML type modifier but the new data type has no type modifier, the existing values do not need to be re-validated. The state of the table space is not changed.

  If the XML schemas that are specified in the old XML type modifier are a subset of the XML schemas that are specified in the new XML type modifier, the existing values do not need to be re-validated. The state of the XML table space is not changed.
- If the XML schemas that are specified in the old XML type modifier are NOT a subset of the XML schemas that are specified in the new XML type modifier, the XML table space for the column that is being changed is left in the CHECK-pending status.

Changing an XML column to use a different type modifier does not result in the invalidation of dependent packages or statements in the dynamic statement cache. Also, changing an XML column to use a different type modifier will not generate a new version of the table.

If the data type is a character or graphic string, the new length attribute must be at least as large as the existing length attribute of the column. If the data type is a numeric data type, the specified precision and scale must be at least as large as the existing precision and scale. If a decimal fraction is being converted to floating point, the ALTER statement will fail if there is a unique index or a unique constraint on the column.

If the specified column has a default value, the existing default value must represent a value that could be assigned to a column with the new data type in accordance with the rules for assignment. The default value is updated to reflect the new data type.

If the column is specified in a unique constraint (unique key or primary key) or unique index, the new column length must not exceed the limit on an index size. For PADDED indexes, the sum of the length attributes of the columns must not be greater than 2000-*n*, where *n* is the number of columns that can contain null values. For NOT PADDED indexes, the sum of the length attributes of the columns must not be greater than 2000-*n*-2*m*, where *n* is the number of nullable columns and *m* is the number of varying length columns.

The total byte count of columns after the alteration must not exceed the maximum row size. If the column is in the partitioning key, the new partitioning key cannot exceed 255-*n*.

Table 100 shows the numeric data type alterations that are supported for SET DATA TYPE:

Table 100. Supported numeric data type alterations for SET DATA TYPE

From/To	SMALLINT	INTEGER	BIGINT	DECIMAL (q,t)	REAL	DOUBLE	DECFLOAT (16)	DECFLOAT (34)
SMALLINT	Y	Y	Y	(q-t)>4	Y	Y	Y	Y
INTEGER	N	Y	Y	(q-t)>9	N	Y	Y	Y
BIGINT	N	N	Y	(q-t)>18	N	N	N	Y
DECIMAL (p,s)	s=0 p<5	s=0 p<10	s=0 p<=19	q>=p (q-t)>=(p-s)	p<7	p<16	p<17	Y
DECFLOAT (16)	N	N	N	N	N	N	Y	Y

Table 100. Supported numeric data type alterations for SET DATA TYPE (continued)

From/To	SMALLINT	INTEGER	BIGINT	DECIMAL (q,t)	REAL	DOUBLE	DECFLOAT (16)	DECFLOAT (34)
DECFLOAT (34)	N	N	N	N	N	N	N	Y
FLOAT (1-21)	N	N	N	N	Y	Y	Y	Y
FLOAT (22-53)	N	N	N	N	N	Y	Y	Y

When a SMALLINT, INTEGER, or DECIMAL column is altered to a BIGINT data type, and there is an index defined on that column, the index will be put in RBDP status.

In releases of DB2 prior to Version 9.1, use of the DECIMAL(19,0) data type for applications that work with BIGINT data was encouraged. For performance reasons, the DECIMAL(19,0) columns should be altered to BIGINT. Note that altering from DECIMAL(19,0) to BIGINT is provided only for DECIMAL(19,0) columns that are used for applications that work with BIGINT (thus, the data in those columns is within the range of the BIGINT).

When altering from DECIMAL(19,0) to BIGINT you should ensure that all values in the DECIMAL(19,0) column are within the range of BIGINT before the alter. The following query or a similar query can be run to determine which rows (if any) contain values that are outside of the range of BIGINT:

```
SELECT * FROM table_name
    WHERE dec19_0_column > 9223372036854775807
    OR dec19_0_column < -9223372036854775808;
```

When a partitioning key column with a numeric data type is altered to a larger numeric data type, and the limit key value for the original numeric data type of the column is X'FF', the limit key value for the new numeric data type of the column is left-padded with X'FF'. For example, if a column is converted from SMALLINT to INTEGER, and a limit key value for the SMALLINT column is 32767 (which is 2 bytes of X'FF'), the limit key for the INTEGER column is 2147483647 (which is 4 bytes of X'FF').

When a partitioning key column with a character data type is altered to a longer character data type, and the limit key value for the original character data type of the column (excluding the first NULL byte if the column is nullable) is neither all X'FF' nor all X'00', the limit key value for the new character data type of the column is right-padded with blank(s) of the encoding scheme of the table. For example, if a column is converted from CHAR(1) to VARCHAR(2), and a limit key value for the CHAR(1) column is 'A' (which is X'C1'), the limit key for the VARCHAR(2) column is 'A ' (which is X'C140' when the encoding scheme of the table is EBCDIC, or is X'C120' when the encoding scheme of the table is UNICODE or ASCII).

When a partitioning key column with a character data type is altered to a longer character data type, and the limit key value for the original character data type of the column (excluding the first NULL byte if the column is nullable) is all X'FF', the limit key value for the new character data type of the column is right-padded with X'FF' and the table space that contains the table being altered is left in REORG-pending (REORP) status.

When a partitioning key column with a character data type is altered to a longer character data type, and the limit key value for the original character data type of the column (excluding the first NULL byte if the column is nullable) is all X'00', the limit key value for the new character data type of the column is right-padded with X'00' and the table space that contains the table being altered is left in REORG-pending (REORP) status.

Table 101 shows the character data type alterations that are supported for SET DATA TYPE:

Table 101. Supported character data type alterations for SET DATA TYPE (x > =0).

From/To	CHARACTER (n+x)	VARCHAR (n+x)	LONG VARCHAR	GRAPHIC (n+x)	VARGRAPHIC (n+x)	LONG VARGRAPHIC
CHARACTER(n)	Y	Y	N	N	N	N
VARCHAR(n)	Y	Y	N	N	N	N
LONG VARCHAR	N	Y	N	N	N	N
GRAPHIC(n)	N	N	N	Y	Y	N
VARGRAPHIC(n)	N	N	N	Y	Y	N
LONG VARGRAPHIC	N	N	N	N	Y	N

When columns are converted from CHAR to VARCHAR, normal assignment rules apply, which means that trailing blanks are kept instead of being stripped out. If you want varying length character strings without trailing blanks, use the STRIP function for data in the column after changing the data type to VARCHAR.

When a CHAR FOR BIT DATA column is converted to a BINARY data type, the following applies:

- The existing space characters in the table will not be changed to hexadecimal zeros (X'00')
- If the new length attribute is greater than current length attribute of the column, the values in the table are padded with hexadecimal zeros (X'00')

When a CHAR FOR BIT DATA or VARCHAR FOR BIT DATA column is converted to a BINARY or VARBINARY data type, the existing default value will be cast as a binary string. The resulting binary string will be at least twice the original size. The alter will fail if the resulting binary string length exceeds 1536 UTF-8 bytes.

When a CHAR FOR BIT DATA or VARCHAR FOR BIT DATA column is converted to a BINARY or VARBINARY data type, and there is an index defined on that column, the index will be put in RBDP.

Table 102. Supported binary data type alterations for SET DATA TYPE (x >= 0)

From/To	BINARY(n+x)	VARBINARY(n+x)
CHAR(n) FOR BIT DATA	Y	Y
VARCHAR(n) FOR BIT DATA	Y	Y
BINARY(n)	Y	Y
VARBINARY(n)	Y[1]	Y

Note: ALTER from VARBINARY to BINARY is not allowed when the column is part of a unique index.

The table being altered must not be defined with an edit procedure that is defined as WITH ROW ATTRIBUTES or a valid procedure. There must not be a materialized query table defined on this table, and this table must not be defined as a materialized query table.

Changing the data type, precision, scale, or length of a column can affect a row permission or a column mask that is defined on the table. If the data type, length, precision, or scale for the column is changed and a column mask is defined for this column, or a row permission or a column mask references this column, these row permissions and column masks are reevaluated using the new column attributes of the column. If an error is encountered during the reevaluation process, the ALTER statement returns the error.

During the reevaluation of the column mask or row permission, user-defined functions that are referenced in the definition of the column mask or the row permission must be resolved to the same functions that were resolved during the creation of the column mask or the row permission.

If the alteration results in the generation of a new table version, the table space that contains the table that is being changed is left in an advisory REORG-pending (AREO) status. If the column that is being changed is part of an index, an exception state might be set for the index as shown in Table 103:

Table 103. Informational settings for ALTER COLUMN when the column is in an index

Alteration type	Exception state for index	Package invalidation
VARCHAR to CHAR	PSRBD	Yes
VARGRAPHIC to GRAPHIC	PSRBD	Yes
CHAR to VARCHAR	AREO*	Yes
GRAPHIC to VARGRAPHIC	AREO*	Yes
VARCHAR to VARCHAR	AREO* (for padded only)	No
VARGRAPHIC to VARGRAPHIC	AREO* (for padded only)	No
CHAR to CHAR	AREO*	Yes
GRAPHIC to GRAPHIC	AREO*	Yes
DECIMAL to DECIMAL	RBDP	Yes
TIMESTAMP WITHOUT TIME ZONE to TIMESTAMP WITHOUT TIME ZONE	AREO*	Yes
TIMESTAMP WITH TIME ZONE to TIMESTAMP WITH TIME ZONE	AREO*	Yes

For information on resetting informational or restrictive exception states, see *DB2 Utility Guide and Reference*.

FOR *subtype* **DATA**
Alters the *subtype* of a character column. This clause does not change the data. The clause only updates the definition of the subtype as it is stored in the DB2 catalog and the OBD. The length and data type that are specified must match the existing length and data type of the column.

Only character strings are valid when subtype is BIT.

For more information on the subtype values (SBCS, MIXED, and BIT), see the subtype information under built-in-type.

SET INLINE LENGTH *integer*

Specifies the new inline length for the column. **SET INLINE LENGTH** can only be specified for an inline LOB column in a table that is in a universal table space. INLINE LENGTH cannot be specified if FOR SBCS DATA or FOR MIXED DATA is also specified in the same ALTER TABLE statement. Inline LOB columns cannot be added to a table that is in a table space that has basic row format. The new length can be smaller or larger than the original length. *integer* is a value between 0 and 32680 bytes (inclusive) for a BLOB or CLOB column or between 0 and 16340 characters (inclusive) for a DBCLOB column. The inline length cannot be changed in the following cases:

- The LOB column is referenced in an expression-based index or a spatial index.
- The new inline length is less than the default length for the column.
- The new inline length is greater than the maximum length of the LOB column.

If there are views that inherit the inline length from a LOB column and the inline length of that LOB column has been changed, the views (and any views that are dependent on those views) are recalculated to use the updated inline length.

If the inline length is changed, any packages or statements in the dynamic statement cache that reference the table are invalidated.

When the base table space is not empty, increasing the length puts the table space in an advisory REORG-pending state, and decreasing the length puts the table space in a REORG-pending state.

No expression-based indexes can be created after the inline length is changed until the REORG utility is run on the base table space.

SET *default-clause*

Specifies the new default value of the column to be altered. The new default value must conform to the current rules for assigning that value to the column. Existing rows will retain their current value. The new default value will only be reflected in the rows that are inserted after the alter. Sections that are dependent on the table that is being altered will be invalidated.

The table must not be referenced by an view. The table must not be defined with the DATA CAPTURE CHANGES attribute when the subsystem parameter RESTRICT_ALT_COL_FOR_DCC is set to YES.

If the column is specified in a unique constraint (unique key or primary key) or unique index, the default value might be altered to the same value as an existing row of that column. However, subsequent data change operations will fail in the absence of a value specified for that column on the insert operation.

For LOB columns, only the default for inline LOB columns can be changed. The new default length cannot be greater than the inline length.

DROP DEFAULT

Drops the current default value of the column. For columns that are not nullable, the specified column must be defined with a default value. For

columns that are nullable, the specified column cannot have a null default value. For columns that are nullable, the new default value is the null value.

The table that contains the specified column must not be referenced in a view. The table must not be defined with the DATA CAPTURE CHANGES attribute when the subsystem parameter RESTRICT_ALT_COL_FOR_DCC is set to YES.

Follow these steps to remove the default value for a column that was defined using ALTER TABLE with the ADD COLUMN clause:

1. Run the REORG utility or the UPDATE statement to reset the AREO* state:
   - Run the REORG utility on the table space that contains the table
   - If the table is in a universal table space and the table does not have row access control activated, run an UPDATE statement without the SKIP LOCKED DATA or WHERE clauses specified. The update operation must be done with a searched UPDATE statement and the expression in the SET clause cannot be a *scalar-fullselect* or a *row-fullselect*. An update operation within a SELECT statement will not reset the AREO* status.
2. Issue the ALTER TABLE statement that specifies the DROP DEFAULT clause

If the REORG is not done before the ALTER TABLE, or the UPDATE statement does not reset the AREO* statue, an error is returned for the ALTER TABLE statement.

SET GENERATED
Specifies that DB2 generates values for the column. SET GENERATED must not be specified for a column of a history table or for a column that already has the GENERATED attribute.

ALWAYS
Specifies that DB2 always generates a value for the column when a row is inserted or updated and a default value must be generated.

BY DEFAULT
Specifies that DB2 generates a value for the column when a row is inserted or updated and a default value must be generated, unless an explicit value is specified. For a row change timestamp column, DB2 inserts or updates a specified value but does not verify that it is a unique value for the column unless the row change timestamp column has a unique constraint or a unique index that solely specifies the row change timestamp column.

RESTART
Specifies the next value for the identity column, If *numeric-constant* is not specified, the sequence is restarted at the value that is specified implicitly or explicitly as the starting value when the identity column was originally created.

WITH *numeric-constant*
Specifies that, when it is time to generate the next value for this identity column, *numeric-constant* will be used as the next value for the column. This value can be any positive or negative value (including 0) that could be assigned to this column without nonzero digits existing to the right of the decimal point.

If RESTART is not specified, the sequence is not restarted. Instead, it resumes with the current values that are in effect for all the options after the ALTER statement is issued.

After an identity column is restarted or changed to allow cycling, sequence numbers might be duplicates of values generated previously.

SET INCREMENT BY *numeric-constant*
For a definition, see the description of INCREMENT BY numeric-constant for defining an identity column.

SET MINVALUE or NO MINVALUE
For a definition, see the description of MINVALUE or NO MINVALUE for defining an identity column.

SET MAXVALUE or NO MAXVALUE
For a definition, see the description of MAXVALUE or NO MAXVALUE for defining an identity column.

SET CYCLE or NO CYCLE
For a definition, see the description of CYCLE or NO CYCLE for defining an identity column.

SET CACHE or NO CACHE
For a definition, see the description of CACHE or NO CACHE for defining an identity column.

SET ORDER or NO ORDER
For a definition, see the description of ORDER or NO ORDER for defining an identity column.

RENAME COLUMN:

RENAME COLUMN *source-column-name* **TO** *target-column-name*
Renames the specified column. The names must not be qualified.

source-column-name
Identifies the column that is to be renamed. The name must identify an existing column of the table.

target-column-name
Specifies the new name for the column. The name must not identify a column that already exists in the table, or the name of a period that exists in the table.

You cannot rename a column if any of the following conditions apply:

- The column is referenced in a view
- The column is referenced in the expression of an index definition
- The column is referenced in the definition of a row permission or a column mask
- The column is referenced in an SQL table user-defined function
- The column has a check constraint defined
- The column has a field procedure defined
- The table has a trigger
- The table is a materialized query table or is referenced by a materialized query table
- The table has a valid procedure, or an edit procedure that is defined as WITH ROW ATTRIBUTES

- The table is a DB2 catalog table
- The table is a system-period temporal table or a history table
- The table is an archive-enabled table or an archive table

DROP COLUMN

DROP COLUMN *column-name*

Drops the identified column from the table. Any privileges that are associated with the column are revoked.

A column cannot be dropped if any of the following conditions are true:

- The containing table space is not a universal table space
- The table is a created global temporary table
- The table is a system-period temporal table
- The table is a history table
- The table is an archive-enabled table
- The table is an archive table
- The table has an edit procedure or a validation exit procedure
- The table contains check constraints
- The table is a materialized query table
- The table is referenced in a materialized query table definition
- The column is defined as a security label column
- The column is an XML column
- The column is a DOCID column
- The column is a hidden ROWID column
- The column is defined as ROWID GENERATED BY DEFAULT, and the table contains a hidden ROWID column
- The column is a ROWID column on which there is a dependent LOB column
- The column is part of the table partitioning key
- The column is part of the hash key
- All of the remaining columns in the table are hidden
- A view that is dependent on the table has INSTEAD OF triggers
- A trigger is defined on the table
- Any of the following objects are dependent on the table:
  - Extended indexes
  - Row permissions
  - Column masks
  - Inline SQL table functions

column-name

Identifies the column that is to be dropped. The column name must not be qualified. The name must identify a column of the specified table. The name must not identify the only column of the table or a column that is referenced in the definition of a period. The table definition must not be in an incomplete state.

If the column is a LOB column, any auxiliary tables that are associated with the column and the indexes on the auxiliary tables are also dropped. Any LOB table spaces that were implicitly created for the auxiliary tables are also dropped. If the column is the last LOB column in the table, any implicitly created ROWID column in the table is also dropped.

Dropping a column is a pending change to the definition of the table if the data sets of the table space are already created, otherwise, the change takes effect immediately.

If the change is a pending change to the definition of the table, the definition of the containing table space must not be in an incomplete state. Pending changes are not reflected in the definition or data at the time the ALTER TABLE statement is issued. Instead, the entire table space is placed in an advisory REORG-pending state (AREOR). A subsequent reorganization of the entire table space will apply the pending definition changes to the definition and data of the table. If the change is a pending change, a new table version is generated.

RESTRICT
Specifies that the column cannot be dropped if any views, indexes, unique constraints, or referential constraints are dependent on the column.

ADD PERIOD:

ADD PERIOD SYSTEM_TIME or BUSINESS_TIME (*start-column-name*, *end-column-name*)
Adds a period to the table.

The table must not be an archive-enabled table or an archive table.

start-column-name must not be the same as *end-column-name*. The data type, precision, and scale for *start-column-name* must be the same as for *end-column-name*.

SYSTEM_TIME
Names the period **SYSTEM_TIME**. The name must not identify an existing column in the table. A table can have only one **SYSTEM_TIME** period.

A system generated check constraint named DB2_GENERATED_CHECK_CONSTRAINT_FOR_SYSTEM_TIME is generated to ensure that the value for *end-column-name* is greater than the value for *start-column-name*. *start-column-name* and *end-column-name* must be defined as TIMESTAMP(12) WITHOUT TIME ZONE. DB2_GENERATED_CHECK_CONSTRAINT_FOR_SYSTEM_TIME cannot be an existing check constraint.

The *start-column-name* must specify a *row-begin* column and the *end-column-name* must specify a *row-end* column. Both columns must be defined as GENERATED ALWAYS. A column mask or row permission must not be defined for the table.

BUSINESS_TIME
Names the period **BUSINESS_TIME**. The name must not identify an existing column in the table. A table can have only one **BUSINESS_TIME** period.

A system generated check constraint named DB2_GENERATED_CHECK_CONSTRAINT_FOR_BUSINESS_TIME is generated to ensure that the value for *end-column-name* is greater than the value for *start-column-name*. DB2_GENERATED_CHECK_CONSTRAINT_FOR_BUSINESS_TIME cannot be an existing check constraint.

The columns that are specified for *start-column-name* and *end-column-name* must be defined as DATE or TIMESTAMP(6) WITHOUT TIME ZONE, and

must be defined as NOT NULL. The columns that are specified for *start-column-name* and *end-column-name* must not identify a column that is defined with a GENERATED clause.

start-column-name
Identifies the column that records the start value for the period. The name must identify an existing column in the table. *start-column-name* must not be the same as a column that is used in the definition of another period for the table.

end-column-name
Identifies the column that records the end value for the period. The name must identify an existing column in the table. *end-column-name* must not be the same as a column that is used in the definition of another period for the table.

ADD unique-constraint:

CONSTRAINT *constraint-name*
Names the primary key or unique key constraint. If a constraint name is not specified, a unique constraint name is generated. If a name is specified, it must be different from the names of any referential, check, primary key, or unique key constraints previously specified on the table. If the table space is implicitly created, the enforcing primary key and unique key indexes are also implicitly created.

PRIMARY KEY(*column-name***,...)**
Defines a primary key composed of the identified columns. Each column name must be an unqualified name that identifies a column of the table. The same column must not be identified more than one time. The following types of columns cannot be specified in a PRIMARY KEY clause:

- a LOB column
- a ROWID column
- a DECFLOAT column
- an XML column
- a distinct type column that is based on a LOB, ROWID, or DECFLOAT data type
- a row change timestamp column
- a Unicode column in an EBCDIC table

The number of identified columns must not exceed 64. In addition, the sum of the length attributes of the columns must not be greater than 2000 -2*m*, where *m* is the number of varying-length columns in the key. The table must not have a primary key and the identified columns must be defined as NOT NULL.

The set of columns in the primary key cannot be the same as the set of columns of another unique key.

The table must have a unique index with a unique key that is identical to the primary key. The keys are identical only if they have the same number of columns and the *n*th column name of one is the same as the *n*th column name of the other. If the table is in a table space that is implicitly created, and no unique index is defined on the identified columns, DB2 will automatically create a primary index. The privilege set must include the INDEX privilege on the table and the USE privilege on the buffer pool and the storage group. The implicitly created primary key index is owned by the owner of the base table.

The identified columns are defined as the primary key of the table. The description of the index is changed to indicate that it is a primary index. If the table has more than one unique index with a key that is identical to the primary key, the selection of the primary index is arbitrary.

BUSINESS_TIME WITHOUT OVERLAPS
BUSINESS_TIME WITHOUT OVERLAPS can be specified as the last item in the list. If BUSINESS_TIME WITHOUT OVERLAPS is specified, the list must include at least one *column-name* or *key-expression*. When WITHOUT OVERLAPS is specified, the values for the rest of the specified keys are unique with respect to the time for the BUSINESS_TIME period. When BUSINESS_TIME WITHOUT OVERLAPS is specified, the columns of the BUSINESS_TIME period must not be specified as part of the constraint. The specification of BUSINESS_TIME WITHOUT OVERLAPS adds the following to the constraint:

- The end column of the BUSINESS_TIME period in ascending order
- The start column of the BUSINESS_TIME period in ascending order

UNIQUE(*column-name***,...)**
Defines a unique key composed of the identified columns with the specified *constraint-name*. If a *constraint-name* is not specified, a name is generated. Each column name must be an unqualified name that identifies a column of the table. The same column must not be identified more than one time. The following types of columns cannot be specified in a UNIQUE clause:

- a LOB column
- a ROWID column
- a DECFLOAT column
- an XML column
- a distinct type column that is based on a LOB, ROWID, or DECFLOAT data type
- a Unicode column in an EBCDIC table

Each identified column must be defined as NOT NULL. The number of identified columns must not exceed 64. In addition, the sum of the length attributes of the columns must not be greater than 2000 - *n* for padded indexes and 2000 - *n* - 2*m* for nonpadded indexes, where *n* is the number of columns that can contain null values and *m* is the number of varying-length columns in the key.

The set of columns in the unique key cannot be the same as the set of columns of the primary key or another unique key. A unique key is a duplicate if it is the same as the primary key or a previously defined unique key. The specification of a duplicate unique key is ignored with a warning.

The table must have a unique index with a key that is identical to the unique key. The keys are identical only if they have the same number of columns and the *n*th column name of one is the same as the *n*th column name of the other. If the table is in a table space that is implicitly created, and no unique index is defined on the identified columns, DB2 will automatically create a unique index to enforce the unique key constraint. The privilege set must include the INDEX privilege on the table and the USE privilege on the buffer pool and the storage group. The implicitly created unique key index is owned by the owner of the base table.

The identified columns are defined as a unique key of the table. The description of the index is changed to indicate that it is enforcing a unique key

constraint. If the table has more than one unique index with a key that is identical to the unique key, the selection of the enforcing index is arbitrary.

BUSINESS_TIME WITHOUT OVERLAPS
BUSINESS_TIME WITHOUT OVERLAPS can be specified as the last item in the list. If BUSINESS_TIME WITHOUT OVERLAPS is specified, the list must include at least one *column-name* or *key-expression*. When WITHOUT OVERLAPS is specified, the values for the rest of the specified keys are unique with respect to the time for the BUSINESS_TIME period. When BUSINESS_TIME WITHOUT OVERLAPS is specified, the columns of the BUSINESS_TIME period must not be specified as part of the constraint. The specification of BUSINESS_TIME WITHOUT OVERLAPS adds the following to the constraint:

- The end column of the BUSINESS_TIME period in ascending order
- The start column of the BUSINESS_TIME period in ascending order

ADD referential-constraint:

CONSTRAINT *constraint-name*
Names the referential constraint. If a constraint name is not specified, a unique constraint name is generated. If a name is specified, it must be different from the names of any referential, check, primary key, or unique key constraints previously specified on the table.

FOREIGN KEY (*column-name,...***)** *references-clause*
Specifies a referential constraint with the specified *constraint-name*.

FOREIGN KEY cannot be specified if the table is a history table or an archive table.

Let T1 denote the object table of the ALTER TABLE statement.

The foreign key of the referential constraint is composed of the identified columns. Each *column-name* must be an unqualified name that identifies a column of T1. The same column must not be identified more than one time. The following types of columns cannot be specified in the FOREIGN KEY clause:

- a LOB column
- a ROWID column
- a DECFLOAT column
- an XML column
- a distinct type column that is based on a LOB, ROWID, or DECFLOAT data type
- a security label column
- a row change timestamp column
- a Unicode column in an EBCDIC table

The number of identified columns must not exceed 64 and the sum of their length attributes must not exceed 255 minus the number of columns that allow null values. The referential constraint is a duplicate if the FOREIGN KEY and the parent table are the same as the FOREIGN KEY and parent table of an existing referential constraint on T1. The specification of a duplicate referential constraint is ignored with a warning.

The foreign key of the referential constraint cannot reference a parent key that contains BUSINESS_TIME WITHOUT OVERLAPS.

REFERENCES *table-name (column-name,...)*

The table name specified after REFERENCES must identify a table that exists at the current server.

table-name must not identify a catalog table, a declared global temporary table, a history table, or an archive table.

Let T2 denote the identified parent table and let T1 denote the table that is being changed (T1 and T2 can be the same table).

T2 must have a unique index and the privilege set on T2 must include the ALTER or REFERENCES privilege on the parent table, or the REFERENCES privilege on the columns of the nominated parent key.

The parent key of the referential constraint is composed of the identified columns. Each *column-name* must be an unqualified name that identifies a column of T2. The same column must not be identified more than one time. The following types of columns cannot be specified in a REFERENCES clause:

- a LOB column
- a ROWID column
- a DECFLOAT column
- an XML column
- a distinct type column that is based on a LOB, ROWID, or DECFLOAT data type
- a security label column
- a row change timestamp column
- a Unicode column in an EBCDIC table

The list of column names in the parent key must be identical to the list of column names in a primary key or unique key in the parent table T2. The column names must be specified in the *same order* as in the primary key or unique key. If any of the referenced columns in T2 has a non-numeric data type, T2 and T1 must use the same encoding scheme.

If a list of column names is not specified, then T2 must have a primary key. Omission of a list of column names is an implicit specification of the columns of the primary key for T2.

The specified foreign key must have the same number of columns as the parent key of T2 and, except for their names, default values, null attributes and check constraints, the description of the *n*th column of the foreign key must be identical to the description of the *n*th column of the nominated parent key. If the foreign key includes a column defined as a distinct type, the corresponding column of the nominated parent key must be the same distinct type. If a column of the foreign key has a field procedure, the corresponding column of the nominated parent key must have the same field procedure and an identical field description. A *field description* is a description of the encoded value as it is stored in the database for a column that has been defined to have an associated field procedure.

The table space that contains T1 must be available to DB2. If T1 is populated, its table space is placed in a check pending status. A table in a segmented table space is populated if the table is not empty. A table in a table space that is not segmented is considered populated if the table space has ever contained any records.

The referential constraint specified by the FOREIGN KEY clause defines a relationship in which T2 is the parent and T1 is the dependent. A description of the referential constraint is recorded in the catalog.

ON DELETE

The delete rule of the relationship is determined by the ON DELETE clause. For more on the concepts used here, see "Referential constraints" on page 23.

If T1 and T2 are the same table, CASCADE or NO ACTION must be specified. SET NULL must not be specified unless some column of the foreign key allows null values. Also, SET NULL must not be specified if any nullable column of the foreign key is a column of the key of a partitioning index. The default value for the rule depends on the value of the CURRENT RULES special register when the ALTER TABLE statement is processed. If the value of the register is 'DB2', the delete rule defaults to RESTRICT; if the value is 'SQL', the delete rule defaults to NO ACTION.

The delete rule applies when a row of T2 is the object of a DELETE or propagated delete operation and that row has dependents in T1. Let *p* denote such a row of T2.

- If RESTRICT or NO ACTION is specified, an error occurs and no rows are deleted.
- If CASCADE is specified, the delete operation is propagated to the dependents of *p* in T1.
- If SET NULL is specified, each nullable column of the foreign key of each dependent of *p* in T1 is set to null.

A cycle involving two or more tables must not cause a table to be delete-connected to itself. Thus, if the relationship would form a cycle:

- The referential constraint cannot be defined if each of the existing relationships that would be part of the cycle have a delete rule of CASCADE.
- CASCADE must not be specified if T2 is delete-connected to T1.

If T1 is delete-connected to T2 through multiple paths, those relationships in which T1 is a dependent and which form all or part of those paths must have the same delete rule and it must not be SET NULL. For example, assume that T1 is a dependent of T3 in a relationship with a delete rule of *r* and that one of the following is true:

- T2 and T3 are the same table.
- T2 is a descendent of T3 and the deletion of rows from T3 cascades to T2.
- T2 and T3 are both descendents of the same table and the deletion of rows from that table cascades to both T2 and T3.

In this case, the referential constraint cannot be defined when *r* is SET NULL. When *r* is other than SET NULL, the referential constraint can be defined, but the delete rule that is implicitly or explicitly specified in the FOREIGN KEY clause must be the same as *r*.

ENFORCED or NOT ENFORCED

Indicates whether or not the referential constraint is enforced by DB2 during normal operations, such as insert, update, or delete.

ENFORCED

Specifies that the referential constraint is enforced by DB2 during normal operations (such as data change operations) and that it is guaranteed to be correct. ENFORCED is the default.

NOT ENFORCED
Specifies that the referential constraint is not enforced by DB2 during normal operations (such as data change operations). NOT ENFORCED should only be used when the data that is stored in the table is verified to conform to the constraint by some other method than relying on DB2.

ENABLE QUERY OPTIMIZATION
Specifies that the constraint can be used for query optimization. DB2 uses the information in query optimization using materialized query tables with the assumption that the constraint is correct. This is the default.

ADD check-constraint:

CONSTRAINT *constraint-name*
Names the check constraint. If *constraint-name* is not specified, a unique constraint name is derived from the name of the first column in the *check-condition* specified in the definition of the check constraint. If a name is specified, it must be different from the names of any referential, check, primary key, or unique key constraints previously specified on the table.

CHECK (*check-condition***)**
Defines a check constraint. At any time, *check-condition* must be true or unknown for every row of the table. A *check-condition* can evaluate to unknown if a column that is an operand of the predicate is null. A *check-condition* that evaluates to unknown does not violate the check constraint. A *check-condition* is a search condition, with the following restrictions:

- It can refer only to the columns of table *table-name*.
- The columns cannot be any of the following types of columns:
  - LOB columns
  - ROWID columns
  - DECFLOAT columns
  - XML columns
  - distinct type columns that are based on LOB, ROWID, and DECFLOAT data types
  - security label columns
  - Unicode columns in an EBCDIC table
- It can be up to 7400 bytes long, not including redundant blanks.
- It must not contain any of the following:
  - Subselects
  - Built-in or user-defined functions
  - CAST specifications
  - Cast functions other than those created when the distinct type was created
  - Host variables
  - Global variables
  - Parameter markers
  - Special registers
  - Columns that include a field procedure
  - CASE expressions
  - ROW CHANGE expressions
  - Row expressions
  - DISTINCT predicates
  - GX constants (hexadecimal graphic string constants)
  - Sequence references
  - OLAP specifications

- If a *check-condition* refers to a LOB column (including a distinct type that is based on a LOB), the reference must occur within a LIKE predicate.
- The AND and OR logical operators can be used between predicates. The NOT logical operator cannot be used.
- The first operand of every predicate must be the column name of a column in the table.
- The second operand in the *check-condition* must be either a constant or a column name of a column in the table.
  - If the second operand of a predicate is a constant, and if the constant is:
    - A floating-point number, then the column data type must be floating point.
    - A decimal number, then the column data type must be either floating point or decimal.
    - A big integer number, then the column data type must not be an integer or a small integer
    - An integer number, then the column data type must not be a small integer.
    - A small integer number, then the column data type must be small integer.
    - A decimal constant, then its precision must not be larger than the precision of the column.
  - If the second operand of a predicate is a column, then both columns of the predicate must have:
    - The same data type
    - Identical descriptions with the exception that the specification of the NOT NULL and DEFAULT clauses for the columns can be different, and that string columns with the same data type can have different length attributes

Effects of defining a check constraint on a populated table: When a check constraint is defined on a populated table and the value of the special register CURRENT RULES is 'DB2', the check constraint is not immediately enforced on the table. The check constraint is added to the description of the table, and the table space that contains the table is placed in a check pending status. For a description of the check pending status and the implications for utility operations, see *DB2 Utility Guide and Reference*.

When a check constraint is defined on a populated table and the value of the special register CURRENT RULES is 'STD', the check constraint is checked against all rows of the table. If no violations occur, the check constraint is added to the table. If any rows violate the new check constraint, an error occurs and the description of the table is unchanged.

DROP constraint:

DROP PRIMARY KEY

Drops the definition of the primary key and all referential constraints in which the primary key is a parent key. The table must have a primary key and the privilege set must include the ALTER or REFERENCES privilege on every dependent table of the table.

The description of the primary index is changed to indicate that it is not a primary index. If the table space was implicitly created, the corresponding enforcing index is dropped if the primary key is dropped.

DROP UNIQUE *constraint-name*

Drops the definition of the unique key constraint and all referential constraints in which the unique key is a parent key. The table must have a unique key.

The privilege set must include the ALTER or REFERENCES privilege on every dependent table of the table. The description of the enforcing index is changed to indicate that it is not enforcing a unique key constraint. If the table space is implicitly created, the corresponding enforcing index is dropped if the unique key is dropped.

DROP FOREIGN KEY *constraint-name*
Drops the referential constraint *constraint-name*. The *constraint-name* must identify a referential constraint in which the table is the dependent table, and the privilege set must include the ALTER or REFERENCES privilege on the parent table of that relationship, or the REFERENCES privilege on the columns of the parent table of that relationship.

DROP CHECK *constraint-name*
Drops the check constraint *constraint-name*. The *constraint-name* must identify an existing check constraint defined on the table.

DROP CONSTRAINT *constraint-name*
Drops the constraint *constraint-name*. The *constraint-name* must identify an existing primary key, unique key, check, or referential constraint defined on the table.

DROP CONSTRAINT must not be used on the same ALTER TABLE statement as DROP PRIMARY KEY, DROP UNIQUE KEY, DROP FOREIGN KEY or DROP CHECK.

ADD partitioning:

ADD PARTITION BY RANGE
Specifies the range partitioning scheme for the table (the columns used to partition the data). When this clause is specified, the table uses table-controlled partitioning. The number of partitions specified in the ADD PARTITION BY RANGE clause has to be the same as the number of partitions defined in the table space.

This clause applies only to tables in a partitioned table space. If the table is already complete by having established either table-controlled partitioning or index-controlled partitioning, the ADD PARTITION BY RANGE clause is not allowed. If this clause is used, then the ENDING AT clause cannot be used on a subsequent CREATE INDEX statement for this table.

partition-expression
Specifies the key data over which the range is defined to determine the target data partition of the data.

column-name
Specifies the columns of the key. Each *column-name* must identify a column of the table. Do not specify more than 64 columns, the same column more than one time, a qualified column name, or any of the following types of columns:

- a BINARY or VARBINARY column
- a LOB column
- DECFLOAT column
- an XML column
- a column with a distinct type that is based on any of the preceding data types
- a row change timestamp column
- a Unicode column in an EBCDIC table

The sum of length attributes of the columns must not be greater than 255 - *n*, where *n* is the number of columns that can contain null values.

A timestamp with time zone column (or a column with a distinct type that is based on the timestamp with time zone data type) can only be specified as the last column in a partitioning key.

NULLS LAST
Specifies that null values are treated as positive infinity for purposes of comparison.

ASC
Puts the entries in ascending order by the column. ASC is the default.

DESC
Puts the entries in descending order by the column.

partition-element
Specifies ranges for a data partitioning key and the table space where rows of the table in the range will be stored.

PARTITION *integer*
Specifies a number of a physical partition in the table space. A PARTITION clause must be specified for every partition of the table space. In the context, highest means highest in the sorting sequence of the columns. In a column that is defined as ascending (ASC), highest and lowest have the usual meanings. In a column that is defined as descending (DESC), the lowest actual value is the highest in the sorting sequence.

ENDING AT (*constant***, MAXVALUE, or MINVALUE...)**
Specifies the limit key for a partition boundary. Specify at least one value (constant, MAXVALUE, or MINVALUE) after ENDING AT in each PARTITION clause. You can use as many values as there are columns in the key. The concatenation of all the values is the highest value of the key for ascending and the lowest for descending.

constant
Specifies a constant value with a data type that must conform to the rules for assigning that value to the column. If a string constant is longer or shorter than required by the length attribute of its column, the constant is either truncated or padded on the right to the required length. If the column is ascending, the padding character is X'FF'. If the column is descending, the padding character is X'00'. The precision and scale of a decimal constant must not be greater than the precision and scale of its corresponding column. A hexadecimal string constant (GX) cannot be specified.

MAXVALUE
Specifies a value greater than the maximum value for the limit key of a partition boundary (that is, all X'FF' regardless of whether the column is ascending or descending). If all of the columns in the partitioning key are ascending, a constant or the MINVALUE clause cannot be specified following MAXVALUE. After MAXVALUE is specified, all subsequent columns must specify MAXVALUE.

MINVALUE
Specifies a value that is smaller than the minimum value for the

limit key of a partition boundary (that is, all X'00' regardless of whether the column is ascending or descending). If all of the columns in the partitioning key are descending, a constant or the MAXVALUE clause cannot be specified following MINVALUE. After MINVALUE is specified, all subsequent columns must be MINVALUE.

The key values are subject to the rules listed for the ENDING AT clause for a partition definition. See list of rules.

INCLUSIVE
Specifies that the specified range values are included in the data partition.

HASH SPACE *integer***K|M|G**
Specifies the amount of fixed hash space to preallocate for the partition that is associated with the partition element. If HASH SPACE is omitted from the partition element, the HASH SPACE value that is specified in the ORGANIZE BY CLAUSE is used.

The HASH SPACE keyword in the *partition-element* must only be specified if the table is defined to use hash organization.

K Indicates that the integer value is to be multiplied by 1024 to specify the hash space size in bytes. The integer must be between 256 and 268435456.

M Indicates that the integer value is to be multiplied by 1048576 to specify the hash space size in bytes. The integer must be between 1 and 262144.

G Indicates that the integer value is to be multiplied by 1073741824 to specify the hash space size in bytes. The integer must be between 1 and 256 for a partition by range table and must be between 1 and 131072 for a non-partitioned table.

If a value greater than 4G is specified, the data sets for the table space are associated with a DFSMS data class that has been specified with extended format and extended addressability.

ADD PARTITION:

ADD PARTITION
Specifies that a partition is added to the table and each partitioned index on the table. The new partition is the next physical partition not being used until the maximum for the table space has been reached. ADD PARTITION must not be specified for nonpartitioned tables. Adding a partition is not allowed if the table is a materialized query table or a materialized query table is defined on the table. However, adding a partition is allowed if an accelerated query table is defined on the table. A partition cannot be added if the table space definition is incomplete because a partitioning key or partitioning index is missing. If the table uses index-controlled partitioning, it is converted to use table-controlled partitioning.

If the table is in a partition-by-growth table space, a new partition can be added until the number of partitions reaches the MAXPARTITIONS limit. The total number of table space partitions cannot exceed the value that is specified for MAXPARTITIONS for the table space.

The maximum number of partitions allowed depends on how the table space was originally created. If DSSIZE was specified when the table space was created, it is non-zero in the catalog. The maximum number of partitions allowed is shown in Table 104.

Table 104. Maximum number of partitions allowed

DSSIZE	Page size 4 KB	Page size 8 KB	Page size 16 KB	Page size 32 KB
1GB-4GB	4096	4096	4096	4096
8GB	2048	4096	4096	4096
16GB	1024	2048	4096	4096
32GB	512	1024	2048	4096
64GB	256	512	1024	2048
128GB	128	256	512	1024
256GB	64	128	256	512

If LARGE was specified when the table space was created, the maximum number of partitions is shown in the fourth row of Table 105. For more than 254 partitions when LARGE or DSSIZE is not specified, the maximum number of partitions is determined by the page size of the table space.

Table 105. Maximum number of partitions when DSSIZE = 0

Type of table space	Number of existing partitions	Maximum partitions
non-large	1 to 16	16
non-large	17 to 32	32
non-large	33 to 64	64
large	N/A	4096

The existing table space PRIQTY and SECQTY attributes of the previous logical partition are used for the space attributes of the new partition. For each partitioned index, the existing PRIQTY and SECQTY attributes of the previous partition are used.

To specify specific space attributes for the new partition, use additional ALTER TABLESPACE and ALTER INDEX statements.

HASH SPACE cannot be specified with ADD PARTITION. For partition-by-growth table spaces, the hash space value is not applicable at the partition level. For range-partitioned universal table spaces, the hash space value is inherited from base table.

ENDING AT (*constant***, MAXVALUE, or MINVALUE, ...)**
Specifies the high key limit for the new partition. The new partition's key limit must be higher when partitioning is ascending and lower when it is descending. Specify at least one value (*constant*, MAXVALUE, or MINVALUE) after ENDING AT in the PARTITION clause. You can use as many values as there are columns in the key. The concatenation of all the values is the highest value of the key in the corresponding partition of the index. ENDING AT cannot be specified for a table in a partition-by growth table space, but must be specified if the table is in a range-partitioned table space.

constant
Specifies a constant value with a data type that must conform to the rules for assigning that value to the column. If a string constant is longer or shorter than required by the length attribute of its column, the constant is either truncated or padded on the right to the required length. If the column is ascending, the padding character is X'FF'. If the column is descending, the padding character is X'00'. The precision and scale of a decimal constant must not be greater than the precision and scale of its corresponding column. A hexadecimal string constant (GX) cannot be specified.

MAXVALUE
Specifies a value greater than the maximum value for the limit key of a partition boundary (that is, all X'FF' regardless of whether the column is ascending or descending). If all of the columns in the partitioning key are ascending, a constant or the MINVALUE clause cannot be specified following MAXVALUE. After MAXVALUE is specified, all subsequent columns must specify MAXVALUE.

MINVALUE
Specifies a value that is smaller than the minimum value for the limit key of a partition boundary (that is, all X'00' regardless of whether the column is ascending or descending). If all of the columns in the partitioning key are descending, a constant or the MAXVALUE clause cannot be specified following MINVALUE. After MINVALUE is specified, all subsequent columns must be MINVALUE.

The key values are subject to the following rules:

- The first value corresponds to the first column of the key, the second value to the second column, and so on. Using fewer values than there are columns in the key has the same effect as using the highest or lowest values for the omitted columns, depending on whether they are ascending or descending.
- The highest value of the key in any partition must be lower than the highest value of the key in the next partition.
- The values specified for the last partition are enforced. The value specified for the last partition is the highest value of the key that can be placed in the table. If the limit was not previously enforced, any existing key values that are greater than the value that is specified for the added partition are placed into the discard data set when REORG is run.
- If a key includes a ROWID column or a column with a distinct type that is based on a ROWID data type, 17 bytes of the constant that is specified for the corresponding ROWID column are considered.
- The combination of the number of table space partitions and the corresponding limit key size cannot exceed the number of partitions * (106 + limit key size in bytes) < 65394
- If the concatenation of all the values exceeds 255 bytes, only the first 255 bytes are considered.

INCLUSIVE
Specifies that the specified range values are included in the data partition.

ALTER PARTITION:

ALTER PARTITION
Specifies that the partitioning limit key for the identified partition is to be changed.

This clause applies only to tables in a partitioned table space. ALTER PARTITION must not be specified for a table in a partition-by-growth table space or for tables that have XML columns.

integer
If *integer* is specified, it must be in the range 1 to *n*, where *n* is the number of partitions in the table. When this option is specified for any partition except for the last, both the identified partition and the partition following are placed in advisory REORG-pending (AREOR) status.

ENDING AT (*constant*, MAXVALUE, or MINVALUE...)
Specifies the highest value of the partitioning key for the identified partition.

In this context, highest means highest in the sorting sequences of the columns. In a column defined as ascending (ASC), highest and lowest have their usual meanings. In a column defined as descending (DESC) the lowest actual value is highest in the sorting sequence.

Specify at least one value after ENDING AT in each ALTER PARTITION clause. You can use as many values as there are columns in the key. The concatenation of all the values is the highest value of the key in the corresponding partition. The length of each highest key value (the limit key) is the same as the length of the partitioning key.

constant
Specifies a constant value with a data type that must conform to the rules for assigning that value to the column. If a string constant is longer or shorter than required by the length attribute of its column, the constant is either truncated or padded on the right to the required length. If the column is ascending, the padding character is X'FF'. If the column is descending, the padding character is X'00'. The precision and scale of a decimal constant must not be greater than the precision and scale of its corresponding column. A hexadecimal string constant (GX) cannot be specified.

MAXVALUE
Specifies a value greater than the maximum value for the limit key of a partition boundary (that is, all X'FF' regardless of whether the column is ascending or descending). If all of the columns in the partitioning key are ascending, a constant or the MINVALUE clause cannot be specified following MAXVALUE. After MAXVALUE is specified, all subsequent columns must specify MAXVALUE.

MINVALUE
Specifies a value that is smaller than the minimum value for the limit key of a partition boundary (that is, all X'00' regardless of whether the column is ascending or descending). If all of the columns in the partitioning key are descending, a constant or the MAXVALUE clause cannot be specified following MINVALUE. After MINVALUE is specified, all subsequent columns must be MINVALUE.

The key values are subject to the rules listed for the ENDING AT clause for a partition definition. See list of rules.

The value that is specified must not be equal to or beyond the range of the partition boundaries of the adjacent partitions.

INCLUSIVE
Specifies that the specified range values are included in the data partition.

HASH SPACE *integer***K|M|G**

Specifies the amount of fixed hash space to preallocate for the partition that is associated with the partition element. If HASH SPACE is omitted from the partition element, the HASH SPACE value that is specified in the ORGANIZE BY CLAUSE is used.

The HASH SPACE keyword in the *partition-element* must only be specified if the table is defined to use hash organization.

K Indicates that the integer value is to be multiplied by 1024 to specify the hash space size in bytes. The integer must be between 256 and 268435456.

M Indicates that the integer value is to be multiplied by 1048576 to specify the hash space size in bytes. The integer must be between 1 and 262144.

G Indicates that the integer value is to be multiplied by 1073741824 to specify the hash space size in bytes. The integer must be between 1 and 256 for a partition by range table and must be between 1 and 131072 for a non-partitioned table.

If a value greater than 4G is specified, the data sets for the table space are associated with a DFSMS data class that has been specified with extended format and extended addressability.

If the table uses index-controlled partitioning, it is converted to use table-controlled partitioning. The high limit key for the last partition is set to the highest possible value for ascending key columns or the lowest possible value for descending key columns.

ROTATE PARTITION:

ROTATE PARTITION FIRST or *integer***TO LAST**

Specifies that the first logical partition or the physical partition that corresponds to *integer* is to be rotated to become the last partition. Processing resets the specified partition to empty, and the limit key that is associated with the partition is set to the constant that is specified with the boundary specification clause. For ascending limit keys, the new limit key must be higher than the limit key for the preexisting last logical partition prior to this statement being processed. For descending limit keys, the new limit must be lower than the limit for the preexisting last logical partition prior to this statement being processed.

The table definition must be complete and must contain more than one partition. This clause must be followed by the ENDING AT clause, which specifies the new high key limit for this partition, which is now logically last.

Rotating a partition occurs immediately. If there is a referential constraint with DELETE RESTRICT on the table, the ROTATE might fail. If the table uses index-controlled partitioning, it is converted to use table-controlled partitioning.

After an ALTER TABLE statement with the ROTATE PARTITION clause is run, the RUNSTATS utility or the REORG utility with the STATISTICS option should be run on the table space to ensure effective access paths are available for selection.

If the table has a security label column, the user must have a valid security label to rotate partitions. In addition, if write-down is in effect, the user must have the write-down privilege.

ROTATE PARTITION must not be specified in the following situations:

- The table is a materialized query table or a materialized query table is defined on the table.
- The table is in a partition-by-growth table space.
- The table has XML columns.
- The table is a system-period temporal table or a history table.
- The table is an archive-enabled table or an archive table.

Adding a partition is allowed if an accelerated query table is defined on the table.

integer
Specifies a positive integer that represents a physical partition number as identified by the PARTITION column of the SYSIBM.SYSTABLEPART catalog table. The partition must be a data partition that exists in the table. The partition cannot be the last partition of the table.

ENDING AT (*constant***, MAXVALUE, or MINVALUE...)**
The ENDING AT clause specifies the new high key limit for the existing partition holding the oldest data.

In this context, highest means highest in the sorting sequences of the columns. In a column defined as ascending (ASC), highest and lowest have their usual meanings. In a column defined as descending (DESC) the lowest actual value is highest in the sorting sequence.

Specify at least one value after ENDING AT. You can use as many values as there are columns in the key. The concatenation of all the values is the highest value of the key in the corresponding partition. The length of each highest key value (the limit key) is the same as the length of the partitioning key.

constant
Specifies a constant value with a data type that must conform to the rules for assigning that value to the column. If a string constant is longer or shorter than required by the length attribute of its column, the constant is either truncated or padded on the right to the required length. If the column is ascending, the padding character is X'FF'. If the column is descending, the padding character is X'00'. The precision and scale of a decimal constant must not be greater than the precision and scale of its corresponding column. A hexadecimal string constant (GX) cannot be specified.

MAXVALUE
Specifies a value greater than the maximum value for the limit key of a partition boundary (that is, all X'FF' regardless of whether the column is ascending or descending). If all of the columns in the partitioning key are ascending, a constant or the MINVALUE clause cannot be specified following MAXVALUE. After MAXVALUE is specified, all subsequent columns must specify MAXVALUE.

MINVALUE
Specifies a value that is smaller than the minimum value for the limit key of a partition boundary (that is, all X'00' regardless of whether the column is ascending or descending). If all of the columns in the partitioning key

are descending, a constant or the MAXVALUE clause cannot be specified following MINVALUE. After MINVALUE is specified, all subsequent columns must be MINVALUE.

The key values are subject to the rules listed for the ENDING AT clause for a partition definition. See list of rules.

INCLUSIVE
Specifies that the specified range values are included in the data partition.

RESET
Specifies that the existing data in the first logical partition is deleted. In addition the key entries from the associated physical and logical index partitions are deleted. In a partitioned table with limit values that are in ascending sequence, ALTER TABLE ROTATE PARTITION FIRST TO LAST logically operates as if the partition with the lowest high key limit were dropped and then a new partition was added with the specified high key limit. The new key limit for the partition must be higher than any other partition in the table. For descending limit keys, the rotation operates as the partition with the highest limit values becomes the partition with the lowest limit values.

If the partition contains referential integrity parent relationships, has DATA CAPTURE logging enabled, or has a delete row trigger, then each data row in the partition must be deleted individually. If a table does not have any of these attribute settings, then the data rows are removed by deleting and redefining the underlying data sets.

ADD VERSIONING:

ADD VERSIONING
Specifies that the table is a system-period temporal table.

The table must not already be defined as a system-period temporal table, a history table, an archive-enabled table, or an archive table.

A SYSTEM_TIME period and a *transaction-start-ID* column must be defined for the table. The data type, length, precision, and scale for a *transaction-start-ID* column must be defined the same as the *row-begin* column and *row-end* column of the SYSTEM_TIME period in the table. The table must be the only table in the table space. The table must not be a materialized query table, an incomplete table, an auxiliary table, a table that is involved in a clone relationship, a table that was implicitly created for an XML column, or a table that contains a security label column. ADD VERSIONING must not be specified with other clauses on the ALTER TABLE statement.

The privilege set must include the privileges to issue an ALTER TABLE statement for the associated history table.

Historical versions of the rows in the table are retained by DB2. A system-period temporal table contains extra information that indicates when a row is inserted into the table, and when it is updated or deleted. An associated history table is used to store the historical rows of the table. When data in the system-period temporal table is updated, the previous version of the row is kept in the associated history table. When data in a system-period temporal table is deleted, the last version of the row is inserted into the history table.

References to the table can include a period clause to indicate which versions of the data are returned.

USE HISTORY TABLE *history-table-name*

Specifies a history table in which to keep the historical rows of the system-period temporal table.

history-table-name must identify a table that exists at the current server and must not identify one of the following tables:

- a catalog table
- an existing system-period temporal table
- an existing history table
- an archive-enabled table
- an archive table
- a declared global temporary table
- a created global temporary table
- a materialized query table
- a view
- an auxiliary table
- a table that was implicitly created for an XML column
- a table that is involved in a clone relationship

The history table must be the only table in the table space. The history table must not contain an identity column, a row change timestamp column, a *row-begin* column, a *row-end* column, a *transaction-start-ID* column, a column mask, a row mask, or a security label column. The history table must not include a period and must not have an incomplete table definition.

The encoding scheme and CCSID for the system-period temporal table and identified history table must be the same.

The system-period temporal table and the identified history table must have the same number and order of columns. The following attributes of the corresponding columns of the two tables must be the same:

- name
- data type
- length (excluding inline LOB length), precision, and scale
- subtype and CCSID
- null attribute
- hidden attribute
- field procedure

If a column of the system-period temporal table is defined as ROWID GENERATED ALWAYS, the corresponding history column should be defined as ROWID GENERATED ALWAYS.

If a column of the system-period temporal table is defined as GENERATED ALWAYS FOR EACH ROW ON UPDATE OF ROW CHANGE TIMESTAMP or GENERATED AS IDENTITY, the corresponding column in the history table cannot be defined with a GENERATED attribute.

DROP VERSIONING:

DROP VERSIONING

Specifies that the table is no longer a system-period temporal table. *table-name* must identify a system-period temporal table. Historical data will no longer be recorded and maintained for the table. The definition of the columns and data of the table *table-name* are not changed, but the table is no longer treated as a

system-period temporal table. The SYSTEM_TIME period is retained. The relationship between the system-period temporal table and history table is removed. The history table is not dropped, only the relationship between the two tables is removed. Subsequent queries that reference the table must not specify a SYSTEM_TIME period specification for the table.

Packages and statements in the dynamic statement cache that use the SYSTEM_TIME period are invalidated.

Versioning cannot be dropped if there are any views, materialized query table definitions, or inline SQL table functions that depend on the SYSTEM_TIME period.

DROP VERSIONING must not be specified with any other clauses on the ALTER TABLE statement.

The privilege set must include the privileges to issue an ALTER TABLE statement for the associated history table.

ADD MATERIALIZED QUERY:

ADD MATERIALIZED QUERY *materialized-query-definition*

Changes a base table to a materialized query table. Supplies a definition for a regular table to make it a materialized query table. The table specified by *table-name* and the result columns of the fullselect must not have the following characteristics:

- Be already defined as a materialized query table
- Have any primary keys, unique constraints (unique indexes), referential constraints (foreign keys), check constraints, or triggers defined
- Be referenced in the definition of another materialized query table
- Be directly or indirectly referenced in the *fullselect*
- Be in an incomplete state
- Be a system-period temporal table or a history table
- Be a base table that has been activated for the row access controls or column access controls
- Be a base table for which a row permission or a column mask has been defined
- Be an archive-enabled table or an archive table

If *table-name* does not meet these criteria, an error occurs.

The fullselect must not contain a period specification.

The object that is specified in the FROM clause of the fullselect cannot be a view with columns of length 0.

fullselect

Defines the query on which the table is based. The columns of the existing table must meet the following characteristics:

- Have the same number of columns
- Have exactly the same column definitions
- Have the same column names in the same ordinal positions

The *fullselect* must not directly or indirectly reference a base table that has been activated for the row access controls or column access controls or reference a base table for which a row permission or a column mask has been defined.

The outer SELECT clause of *fullselect* must not result in a column that is an array.

If fullselect is specified, the owner of the table being altered must have the SELECT privilege on the tables or views referenced in the fullselect. Having SELECT privilege means that the owner has at least one of the following authorizations:

- Ownership of the tables or views referenced in the fullselect
- The SELECT privilege on the tables and views referenced in the fullselect
- SYSADM authority
- DBADM authority for the database in which the table of the fullselect reside

If the owner of the table does not have the SELECT privilege, the following authorization IDs must have SYSADM authority or DBADM authority for the database in which the tables of the fullselect reside:

- For embedded statements, the authorization ID of the owner of the plan or package
- For dynamically prepared statements, the SQL authorization ID of the process

For details about specifying *fullselect* for a materialized query table, see the definition of *fullselect* in the "CREATE TABLE" on page 1388 statement.

Altering a table to change it from a base table to a materialized query table with REFRESH DEFERRED causes any packages that are dependent on the table to be invalidated.

refreshable-table-options
Specifies the materialized query table options for altering a regular table to a materialized query table. The ORDER BY clause is allowed, but it is used only by REFRESH. The ORDER BY clause can improve the locality of reference of data in the materialized query table.

DATA INITIALLY DEFERRED
Specifies that the data in the table is not validated as part of the ALTER TABLE statement. A REFRESH TABLE statement can be used to make sure the data in the materialized query table is the same as the result of the query in which the table is based.

REFRESH DEFERRED
Specifies that the data in the table can be refreshed at any time using the REFRESH TABLE statement. The data in the table only reflects the result of the query as a snapshot at the time when the REFRESH TABLE statement is processed or as updated by the user for a user-maintained materialized query table.

MAINTAINED BY SYSTEM or MAINTAINED BY USER
Specifies how the data in the materialized query table is maintained.

MAINTAINED BY SYSTEM
Specifies that the data in the materialized query table *table-name* is to be maintained by the system. Only the REFRESH TABLE statement is allowed on the table.

MAINTAINED BY USER
Specifies that the data in materialized query table *table-name* is to

be maintained by the user, who can use LOAD utility or SQL data change statements and REFRESH TABLE statements on the table.

ENABLE QUERY OPTIMIZATION or DISABLE QUERY OPTIMIZATION
Specifies whether this materialized query table can be used for optimization.

ENABLE QUERY OPTIMIZATION
Specifies that the materialized query table can be used for query optimization. If the fullselect specified does not satisfy the restrictions for query optimization, an error occurs. For detailed rules to satisfy query optimization, see *materialized-query-definition* in the "CREATE TABLE" on page 1388 statement.

DISABLE QUERY OPTIMIZATION
Specifies that the materialized query table cannot be used for query optimization. The table can still be queried directly.

ALTER MATERIALIZED QUERY:

ALTER MATERIALIZED QUERY *materialized-query-table-alteration*
Changes attributes of a materialized query table. The *table-name* must identify a materialized query table.

SET *refreshable-table-alteration*
Changes how the table is maintained or whether the table can be used in query optimization.

MAINTAINED BY SYSTEM
Specifies that the data in a materialized query table *table-name* is to be maintained by the system.

MAINTAINED BY USER
Specifies that the data in the materialized query table *table-name* is to be maintained by the user.

ENABLE QUERY OPTIMIZATION
Specifies that materialized query table *table-name* can be used in query optimization. If the fullselect specified for the materialized query table does not satisfy the restrictions for automatic query optimization, an error occurs. For detailed rules to satisfy query optimization, see "CREATE TABLE" on page 1388.

DISABLE QUERY OPTIMIZATION
Specifies that materialized query table *table-name* cannot be used for query optimization. The table can still be queried directly.

DROP MATERIALIZED QUERY:

DROP MATERIALIZED QUERY
Changes a materialized query table so that it is no longer considered a materialized query table. The table specified by *table-name* must be defined as a materialized query table. The definition of columns and data of the name are not changed, but the table can no longer be used for query optimization and is no longer valid for use with the REFRESH TABLE statement.

Altering a table to change from a materialized query table to a base table with the DROP MATERIALIZED QUERY clause causes any packages dependent on the table to be invalidated.

DATA CAPTURE:

DATA CAPTURE

Specifies whether the logging of the following actions on the table is augmented by additional information:

- SQL data change operations
- Adding columns (using the ADD COLUMN clause)
- Changing columns (using the ALTER COLUMN clause)

For guidance on intended uses of the expanded log records, see:

- The description of data propagation to IMS in *IMS DataPropagator: An Introduction*
- The instructions for using Remote Recovery Data Facility (RRDF) in *Remote Recovery Data Facility Program Description and Operations*
- The instructions for reading log records in *DB2 Administration Guide*

NONE

Do not record additional information to the log.

CHANGES

Write additional data about SQL updates to the log. Information about the values that are represented by any LOB or XML columns is not available. Do not specify DATA CAPTURE CHANGES for tables that reside in table spaces that specify NOT LOGGED.

The DATA CAPTURE CHANGES clause can be specified for a table for which row access controls or column access control are active. However, the access controls do not protect data that is written to the log.

For details about the recording of additional data for logged updates to catalog tables, see "Notes" on page 979.

VOLATILE:

VOLATILE or NOT VOLATILE

Specifies how DB2 is to choose access to the table.

VOLATILE

Specifies that DB2 is to use index access to the table whenever possible for SQL operations. However, be aware that list prefetch and certain other optimization techniques are disabled when VOLATILE is used.

One instance in which you might want to use VOLATILE is for a table whose size can vary greatly. If statistics are taken when the table is empty or has only a few rows, those statistics might not be appropriate when the table has many rows. Another instance in which you might want to use VOLATILE is for a table that contains groups of rows, as defined by the primary key on the table. All but the last column of the primary key of such a table indicate the group to which a given row belongs. The last column of the primary key is the sequence number indicating the order in which the rows are to be read from the group. VOLATILE maximizes concurrency of operations on rows within each group, since rows are usually accessed in the same order for each operation.

NOT VOLATILE

Specifies that DB2 is to base SQL access to the table on the current statistics.

CARDINALITY
An optional keyword that currently has no effect, but that is provided for DB2 family compatibility.

ADD CLONE:

ADD CLONE *clone-table-name*
Specifies that a clone table, identified by *clone-table-name*, is created for the table that is being altered. The name, including the implicit or explicit qualifiers, must not identify a table, view, alias, or synonym that exists at the current server. The name must not identify a table that exists in the SYSPENDINGOBJECTS catalog table. The clone table is created in the same table space as the base table and has the same structure as the base table. This includes, but is not limited to, column names, data types, null attributes, check constraints, indexes. When ADD CLONE is used to create a clone of the specified base table, the base table must conform to the following rules:

- Reside in a DB2-managed universal table space
- If the table space or any of its dependent objects (LOBs, XMLs, or indexes) is created with the DEFINE NO clause, all data sets must already be created
- Be the only table in the table space
- Not be defined with a clone table
- Not be defined to use hash organization.
- Not be involved in any referential constraint
- Not be defined with any after triggers
- Not be a materialized query table
- Not have any pending changes
- Not have any active versioning
- Not have an incomplete definition
- Not be a created global temporary table or a declared global temporary table
- Not be a system-period temporal table or a history table
- Not be an archive-enabled table or an archive table

The base table and the clone table are considered unrelated with regard to access controls. Row access control or column access control can be activated independently for the base table, the clone table, or both.

DROP CLONE:

DROP CLONE
Specifies that the clone table that is associated with the specified base table is dropped. *table-name* must identify a base table that exists at the current server and the table must have a clone table defined.

When a clone table is dropped, any row permissions or column masks that are defined for the clone table are also dropped. If the clone table is referenced in the definition of a row permission or a column mask, the ALTER statement returns an error

RESTRICT ON DROP:

ADD RESTRICT ON DROP
Restricts dropping the table and the database and table space that contain the table.

DROP RESTRICT ON DROP
Removes the restriction on dropping the table and the database and table space that contain the table.

ADD organization:

ADD ORGANIZE BY HASH
Specifies that a hash is to be used for the data organization of the table.

ADD ORGANIZE BY HASH must not be specified if the table is already defined with the APPEND YES clause, or if the table space is defined with the MEMBER CLUSTER clause

ALTER TABLE ADD ORGANIZE BY HASH is allowed only if the table is in either a partition-by-growth table space or a range-partitioned universal table space.

ADD ORGANIZE BY HASH must not be specified on tables that are using basic row format.

ADD ORGANIZE BY HASH must not be specified if a user specified clustering index exists.

ADD ORGANIZE BY HASH must not be specified for global temporary tables.

After ALTER TABLE with ADD ORGANIZE BY HASH runs:

- Any packages that are dependent on the table are invalidated.
- All columns that are part of the hash key are no longer updatable. SQL statements that update a column of the hash key return an error.
- The entire table space that contains the table must be reorganized.

UNIQUE
Specifies that DB2 enforces uniqueness of the hash key columns, preventing the table from containing two or more rows with the same value of the hash key.

(*column-name*,...)
The list of column names defines the hash key that is used to determine where a row will be placed.

Each *column-name* must be an unqualified name that identifies a column of the table. The same column must not be specified more than one time and the specified columns must be defined as NOT NULL. The number of specified columns must not exceed 64, and the sum of their length attributes must not exceed 255. A specified column cannot be any of the following types of columns:

- a LOB column
- a DECFLOAT column
- an XML column
- a distinct type column that is based on one of the preceding data types
- a Unicode column in an EBCDIC table

If the table is defined as partition by range, the list of column names must specify all of the column names that are specified in the *partition-expression* for the table, and must specify the column names in the same order as *partition-expression*. If the ORGANIZE BY clause contains more columns than the *partition-expression* for the table, *partition-expression* determines the partition number.

HASH SPACE *integer***K|M|G**

Specifies the amount of fixed hash space to preallocate for the table. If the table is range-partitioned, this is the space for each partition.

The default is 64M for a table in a partition-by-growth universal table space or 64M for each partition of a partition by range universal table space.

K Indicates that the integer value is to be multiplied by 1024 to specify the hash space size in bytes. The integer must be between 256 and 268435456.

M Indicates that the integer value is to be multiplied by 1048576 to specify the hash space size in bytes. The integer must be between 1 and 262144.

G Indicates that the integer value is to be multiplied by 1073741824 to specify the hash space size in bytes. The integer must be between 1 and 256 for a partition by range table and must be between 1 and 131072 for a non-partitioned table.

If a value greater than 4G is specified, the data sets for the table space are associated with a DFSMS data class that has been specified with extended format and extended addressability.

ALTER ORGANIZATION:

ALTER ORGANIZATION SET HASH SPACE *integer*

Changes the fixed hash space that is used for the data organization for the table. The table must be defined to use hash organization.

If the table is defined as range-partitioned, the value specified by *integer* is per partition and applies to each partition of the table. For tables that are not range-partitioned, *integer* applies to the whole table.

The new hash space value will be applied when the table space is reorganized using the REORG utility.

HASH SPACE *integer***K|M|G**

Specifies the amount of fixed hash space to preallocate for the table. If the table is range-partitioned, this is the space for each partition.

K Indicates that the integer value is to be multiplied by 1,024 to specify the hash space size in bytes. The integer must be between 256 and 67,108,864.

M Indicates that the integer value is to be multiplied by 1,048,576 to specify the hash space size in bytes. The integer must be between 1 and 65,536.

G Indicates that the integer value is to be multiplied by 1,073,741,824 to specify the hash space size in bytes. The integer must be between 1 and 64 for a range-partitioned table and must be between 1 and 131,072 for a non-partitioned table.

If a value greater than 4G is specified, the data sets for the table space are associated with a DFSMS data class that has been specified with extended format and extended addressability.

DROP ORGANIZATION:

DROP ORGANIZATION

Specifies that the data organization definition for the table is dropped. The entire table becomes inaccessible and is placed in REORG-pending status. REORG must be run to make the table accessible. If the table is in a partition by range universal table space, the entire table space must be reorganized at one time.

If any type of clustering is required, you must create the clustering index or add the MEMBER CLUSTER clause to the table.

After the next time the REORG utility is run, the hash space value will be cleared and the implicitly created hash overflow index will be dropped.

DROP ORGANIZATION must only be specified if the table is defined to use hash organization.

To change the columns that are specified for the hash key for a table that uses hash organization, the definition of the hash key must be dropped by using ALTER DROP ORGANIZATION, then the new columns for the hash key can be specified with ALTER ADD *organization-clause*.

ROW ACCESS CONTROL:

ACTIVATE ROW ACCESS CONTROL

Specifies that row access control should be activated for the table. If the table is an alias or a synonym, row access control is activated for the base table.

The table must not be one of the following tables:

- A created temporary table
- A table that is directly or indirectly referenced in the definition of a materialized query table
- A table that has a security label column
- A system-period temporal table
- A history table
- An archive-enabled table
- An archive table

If a trigger exists for the table, the trigger must be defined with the SECURED clause.

The table must not be referenced in the definition of a view if the following conditions are true:

- The view is defined with the WITH CHECK OPTION clause
- An INSTEAD OF trigger exists for the view and the trigger is not defined with the SECURED clause.

A default row permission is implicitly created for the table and allows no access to any of the rows of the table, unless there is another row permission that is enabled and that provides access for the authorization IDs or roles that are specified in the definition of the row permission. A query that references the table before such a row permission exists and is enabled will return a warning that there is no data in the table.

ACTIVATE ROW ACCESS CONTROL must not be specified if a period is defined for the table, because a default row permission cannot be defined for a table with a period specification.

When the table is referenced in a SELECT, INSERT, UPDATE, DELETE, or MERGE statement, all row permissions that are enabled for the table, including the default row permission, are applied to control the set of rows that are accessible for the table. If any row permission that is enable is invalid because a previous attempt to regenerate the row permission was unsuccessful, row access control cannot be activated.

ACTIVATE ROW ACCESS CONTROL is ignored if row access control is already activated for the table.

DEACTIVATE ROW ACCESS CONTROL
Specifies that row access control for the table is deactivated. When the table is referenced in a SELECT, INSERT, UPDATE, DELETE, or MERGE statement, any existing row permissions for the table that are enable are not applied to control the set of rows that are accessible for the table.

DEACTIVATE ROW ACCESS CONTROL is ignored if row access control is already defined as not activated for the table.

COLUMN ACCESS CONTROL:

ACTIVATE COLUMN ACCESS CONTROL
Specifies that column access control should be activated for the table. If the table is an alias or a synonym, column access control is activated for the base table.

The table must not be one of the following tables:

- A created temporary table
- A table that is directly or indirectly referenced in the definition of a materialized query table
- A system-period temporal table
- A history table
- An archive-enabled table
- An archive table

If a trigger exists for the table, the trigger must be defined with the SECURED clause.

The table must not be referenced in the definition of a view if the following conditions are true:

- The view is defined with the WITH CHECK OPTION clause
- An INSTEAD OF trigger exists for the view and the trigger is not defined with the SECURED clause.

When column access control is activated, access to the table is not restricted. However, when the table is referenced in a SELECT, INSERT, UPDATE, DELETE, or MERGE statement, all column masks that are enabled for the table are applied to mask the values that are returned for the columns that are referenced in the final result table or to determine the new values that are used in the SQL data change statements. If any enabled column mask is invalid because a previous attempt to regenerate it was unsuccessful, column access control cannot be activated

ACTIVATE COLUMN ACCESS CONTROL is ignored if column access control is already activated for the table.

DEACTIVATE COLUMN ACCESS CONTROL
Specifies that column access control for the table is deactivated. When the table is referenced in a SELECT, INSERT, UPDATE, DELETE, or MERGE statement,

any existing column masks that are enabled for the table are not applied to control the values that are returned for the columns that are referenced in the final result table or to determine if the new values can be used in the SQL data change statements.

DEACTIVATE COLUMN ACCESS CONTROL is ignored if column access control is already defined as not activated for the table.

APPEND:

APPEND NO or APPEND YES

Specifies whether append processing is used for the table. The APPEND clause must not be specified for a table in a work file table space.

If the base table is in a range-partitioned table space, the APPEND option on the LOB table might be different for each partition (depending if the LOB table space and associated objects for each partition are created explicitly or implicitly). If the base table is in a partition-by-growth table space, the APPEND attributes of LOB table will be inherited by each partition.

NO Specifies that append processing is not used for the table. For insert and LOAD operations, DB2 will attempt to place data rows in a well clustered manner with respect to the value in the row's cluster key columns.

YES

Specifies that data rows are placed into the table without regard to clustering during the insert and LOAD operations.

AUDIT:

AUDIT

Alters the auditing attribute of the table. For information about audit trace classes, see *DB2 Administration Guide*.

NONE

Specifies that no auditing is to be done when the table is accessed.

CHANGES

Specifies that auditing is to be done when the table is accessed during the first insert, update, or delete operation. However, the auditing is done only if the appropriate audit trace class is active.

ALL

Specifies that auditing is to be done when the table is accessed during the first operation of any kind performed by a utility or application process. However, the auditing is done only if the appropriate audit trace class is active and the access is not performed with COPY, RECOVER, REPAIR, or any stand-alone utility.

The ALTER TABLE statement is audited for successful and failed attempts in the following cases, if the appropriate audit trace class is active:

- **AUDIT** attribute is changed to **NONE**, **CHANGES**, or **ALL** on an audited or non-audited table.
- **AUDIT CHANGES** or **AUDIT ALL** is in effect.

VALIDPROC:

VALIDPROC

Names a validation procedure for the table or inhibits the execution of any existing validation procedure.

program-name
Designates *program-name* as the new validation exit routine for the table.

The validation procedure can inhibit a data change operation on any row of the table. Before the operation takes place, the row is passed to the procedure. The values that are represented by any LOB or XML columns in the table are not passed to the validation procedure. On an insert or update operation, if the table has a security label column and the user does not have write-down privilege, the user's security label value is passed to the validation routine as the value of the column. After examining the row, the procedure returns a value that indicates whether the operation should proceed. A typical use is to impose restrictions on the values that can appear in various columns.

A table can have only one validation procedure at a time. When you name a new procedure, any existing procedure is no longer used. The new procedure is not used to validate existing table rows. It is used only to validate rows that are loaded, inserted, updated, or deleted after execution of the ALTER TABLE statement.

The table must not be an EBCDIC table that includes a Unicode column.

Related information:

Validation routines (DB2 Administration Guide)

NULL
Discontinues the use of any validation routine for the table.

ENABLE ARCHIVE:

ENABLE ARCHIVE
Specifies that the table is an archive-enabled table.

The table must satisfy the following criteria:

- The table must not already be defined as an archive-enabled table or an archive table.
- The table must not contain a period.
- The table must be the only table in the table space.
- The table must not have a column mask or row permission defined.
- The table must not be one of the following tables:
  - A materialized query table
  - An incomplete table
  - An auxiliary table
  - A table that is involved in a clone relationship
  - A table that was implicitly created for an XML column
  - A table that contains a security label column
  - A system-period temporal table
  - A history table

ENABLE ARCHIVE must not be specified with other clauses on the ALTER TABLE statement.

The privilege set must include the privileges to issue an ALTER TABLE statement for the associated archive table.

For archive-enabled tables, DB2 retains archived versions of the rows. When data in an archive-enabled table is deleted, and the

SYSIBMADM.MOVE_TO_ARCHIVE global variable is set to Y, the last version of the row is inserted into the archive table.

The SYSIBMADM.GET_ARCHIVE global variable and the ARCHIVESENSITIVE bind option determine whether rows in the associated archive table are included when an archive-enabled table is referenced in a *table-reference*.

Related information:

Archive-enabled tables and archive tables (Introduction to DB2 for z/OS)

"References to built-in global variables" on page 223

ARCHIVESENSITIVE bind option (DB2 Commands)

USE *archive-table-name*

Specifies an archive table in which to keep archived rows of the archive-enabled table.

archive-table-name must identify a table that exists at the current server. The table must satisfy the following criteria:

- The table must be the only table in the table space.
- The table must not have an incomplete table definition.
- The table must not be defined as the parent or child in an existing referential constraint.
- The table must not include a period.
- The table must not include a row permission or column mask.
- The table cannot be one of the following tables:
  - A catalog table
  - An archive-enabled table
  - An existing archive table
  - A system-period temporal table
  - A history table
  - A declared global temporary table
  - A created global temporary table
  - A materialized query table
  - A view
  - An auxiliary table
  - A table that was implicitly created for an XML column
  - A clone table
  - A table that has a clone defined on it
- The table must not contain any of the following columns:
  - An identity column
  - A row-begin column
  - A row-end column
  - A transaction-start-ID column
  - A security label column

The privilege set must include the privileges to issue an ALTER TABLE statement for the associated archive table.

The archive-enabled table and the associated archive table must have the same number and order of columns. The following attributes for the corresponding columns of the two tables must be the same:

- Name
- Data type
- Length (excluding inline LOB length or XML length in the base table), precision, and scale
- FOR BIT, SBCS, or MIXED DATA attribute for character string columns
- Null attribute
- Hidden attribute
- CCSID
- Field procedure

If a column of an archive-enabled table is defined as ROWID, the corresponding column of the archive table must also be defined as ROWID with the GENERATED ALWAYS attribute.

If a column of an archive-enabled table is defined as row change timestamp, the corresponding column of the archive table must also be defined as row change timestamp with the GENERATED ALWAYS attribute.

DISABLE ARCHIVE:

DISABLE ARCHIVE

Specifies that the table is no longer an archive-enabled table.

table-name must identify an archive-enabled table. The definition of the columns and data of the table *table-name* are not changed, but the table is no longer treated as an archive-enabled table. The relationship between the archive-enabled table and the associated archive table is removed. The archive table is not dropped. However, by removing the relationship between the archive table and the archive-enabled table, the behavior of the archive-enabled table changes as follows:

- Subsequent queries that reference the table do not consider rows in the archive table regardless of the setting of the SYSIBMADM.GET_ARCHIVE global variable or the ARCHIVESENSITIVE bind option.
- Deleted rows are not moved to the archive table regardless of the setting of the SYSIBMADM.MOVE_TO_ARCHIVE global variable.

Packages and statements in the dynamic statement cache that reference the table are invalidated.

DISABLE ARCHIVE must not be specified with any other clauses on the ALTER TABLE statement.

The privilege set must include the privileges to issue an ALTER TABLE statement for the associated archive table

Notes®

Order of processing of clauses:

When there is more than one clause, they are processed in the following order:

1. VALIDPROC
2. AUDIT
3. DATA CAPTURE

4. ROTATE
5. VOLATILE clauses
6. APPEND clauses
7. DROP clauses
8. ALTER clauses
9. RENAME clause
10. ADD clauses

Within each of these stages, the order in which the user specifies the clauses is the order in which they are performed.

Altering the data type, length, precision, or scale of a column:

When you change the data type, length, precision, or scale of a column, the following information applies to indexes, limit keys, check constraints, and invalidation:

- *Restrictions.* The ALTER TABLE statement is not allowed if any of the following conditions are true:
  - The column is referenced in a referential constraint.
  - The column has a field procedure routine.
  - The column is defined as an identity column.
  - The column is defined as an security label column
  - The table has an edit or validation routine.
  - The subsystem parameter RESTRICT_ALT_COL_FOR_DCC is set to YES, the table is defined with DATA CAPTURE CHANGES, and the ALTER TABLE statement specifies an ALTER COLUMN clause that attempts to drop the default value for a column.
  - The table is a created temporary table.
  - The table is a materialized query table or the table is referenced by a materialized query table.
  - The data type changed is not to a compatible data type.
  - The new length or data type specification could result in a loss of significance because of a shorter length or less precision in the data type.
  - For a conversion from decimal to float, a unique index or a unique constraint exists on the column.
  - For a conversion from other numeric data type to DECFLOAT, a partitioning key, check constraint, index, or a unique constraint exists on the column.
  - The existing default value for a column cannot be assigned to the new data type.
  - Increasing the column length results in an existing index that references the column exceeding the maximum size of an index.
  - Increasing the column length results in the partitioning key using that column exceeding the maximum size for a partitioning key.
  - Table definition is incomplete because unique index for enforcing a unique constraint (primary key or unique key) is missing.
- *Indexes.*
  - If the index has a changed character column, the index is in advisory REORG-pending (AREO*) status.

- If the index has a changed numeric column, the index remains in REBUILD-pending (RBDP) status.
- If the index has a changed timestamp (with or without time zone) column, the index is in advisory REORG-pending (AREO*) status.

- *Length of partitioned index keys.* When a table is altered and the length of a column in the PARTITIONING KEY is changed, DB2 changes the length of the limit key (the highest key value) for a partition too. The length of the limit key is increased by the same amount that the length of the column is increased.
- *Check constraints.* If a check constraint refers to the column being altered, the length of the column is also changed in the check constraint.
- *Statistics.* The RUNSTATS utility should be run to collect new COLUMN statistics for all altered columns. Even though the COLCARDF value is valid, the HIGH2KEY and LOW2KEY values are invalid, and any SYSCOLSTATS catalog entries for the column are removed. Any frequencies or histogram statistics which include this column should also be collected again.

When you change a column from a fixed to varying length or change the length of a varying-length column, process the ALTER TABLE statements in the same unit of work or do a reorganization between the ALTER TABLE statements to avoid anomalies with the lengths and padding of individual values.

Referencing columns in ADD, ALTER, and RENAME clauses:

A column can only be referenced once in an ADD COLUMN, an ALTER COLUMN, or a RENAME COLUMN clause in a single ALTER TABLE statement. However, that same column can be referenced multiple times for adding or dropping constraints in the same ALTER TABLE statement.

Because a distinct type is subject to the same restrictions as its source type, all the syntactic rules that apply to LOB, ROWID, and DECFLOAT columns apply to distinct type columns that are based on LOBs, row IDs, and DECFLOATs. For example, if a table has an explicitly created ROWID column, you cannot add a column with a distinct type that is sourced on a row ID.

Adding a column to table T only changes the description of T. If the catalog description of T is used to create a table T' and a facility such as DSN1COPY is used to effectively copy T into T', queries that refer to the added column in T' will fail because the data does not match its description. To avoid this problem, run the REORG utility against the table space of T before making the copy.

Restrictions on a clone table:

Tables that are involved in a clone relationship (base tables and their associated clone tables) have the following restrictions:

- You cannot use the RUNSTATS utility on a clone table.
- Objects that are involved in a clone relationship do not use the FASTSWITCH naming convention when the REORG utility is run. This includes both the base table and the clone table objects (data and index), as well as LOB and XML objects.
- For a partitioned table, if a mixture of 'I' and 'J' data sets exists when a clone table is created, the mixture of 'I' and 'J' data sets can be changed only by first dropping the clone table.
- Catalog and directory tables cannot have clone tables.

- Indexes cannot be created on a clone table. When an index is created on a base table that is involved in a clone relationship, the index on the clone table will be created implicitly and will be put into rebuild-pending status.
- Implicitly created auxiliary table spaces (table spaces for LOB and XML columns) and auxiliary indexes for the base table are always created as DEFINE YES.
- Before triggers cannot be created on a clone table. Before triggers that are created on a base table apply to both the base table and the clone table.
- You cannot rename a base table that has a clone and you cannot rename a clone table.
- Real-time statistics tables cannot have clone tables.
- You cannot drop an auxiliary table or an auxiliary index of an object that is involved in a clone relationship.

If the table is involved in a clone relationship, no other table altering can take place. If a table change is required, the clone table objects must be dropped so that the base table object attributes can be modified. After the table and index changes and such are completed, the clone table objects can be recreated.

Size restriction for the object descriptor of a table in the SYSIBM.SYSOBDS catalog table:

The following cases might result in an error being returned if the ALTER TABLE statement results in a versioned object descriptor that is larger than 30,000 bytes being added (or updated) in the SYSIBM.SYSOBDS catalog table:

- An ALTER TABLE statement that results in the first version of the object descriptor being generated for the table
- An ALTER TABLE statement that results in the first version of the object descriptor being generated for one or more of the indexes that are defined on the table
- An ALTER TABLE ALTER COLUMN SET DATA TYPE statement on an existing decimal column on a versioned table

You might need to drop and recreate the table if the object descriptor for the table exceeds 30,000 bytes. Alternatively, you can reduce the size of the object descriptor for the table by reducing the size of the default value for varying-length columns in the table by issuing an ALTER TABLE ALTER COLUMN SET DEFAULT statement. You can also drop unnecessary column defaults to reduce the size of the object descriptor for the table.

Altering the attributes of an existing identity column:

Existing values for the identity column are unaffected by the ALTER TABLE statement. The changed identity column attributes affect values generated after the ALTER statement has executed. DB2 does not validate any of the existing identity column values against the new identity column attributes. For example, duplicate values might be generated even if NO CYCLE is in effect, such as when an ascending identity column altered to become a descending identity column.

Any existing values in the cache that have not yet been used might be lost. Loss of cached values can also occur if the ALTER statement returns an error or is rolled back.

Pending changes to the definition of a table:

Issuing the ALTER TABLE statement with certain options can cause a pending change to the definition of a table. When an ALTER TABLE statement that causes pending changes to the definition is executed, semantic validation and authorization checking are performed. However, changes to the table definition and data are not applied and the table space is placed in advisory REORG-pending state (AREOR). The pending changes are recorded in the SYSIBM.SYSPENDINGDDL catalog table. Run the REORG utility with the SHRLEVEL CHANGE or SHRLEVEL REFERENCE options on the table space to apply the pending changes to the definition and data of the table space. When the pending changes are applied, dependent packages are invalidated, the corresponding entries in the SYSIBM.SYSPENDINGDDL catalog table are removed, and the advisory REORG-pending state is removed.

The following ALTER TABLE options can cause pending changes to the definition of the table under certain conditions:

- DROP COLUMN, if the data sets of the table space are already created
- ALTER PARTITION, to change the limit keys for the following types of partitioned table spaces:
  - Range-partitioned universal table spaces
  - Partitioned table spaces (non-universal) with table-controlled partitioning

Restrictions when objects have pending definition changes:

The following statements cannot be executed if the table space, or any objects within that table space, has pending definition changes:

- ALTER TABLE with immediate options
- CREATE INDEX on the table
- ALTER INDEX ADD COLUMN or ADD INCLUDE COLUMN of any index defined on the table
- ALTER INDEX REGENERATE of any index defined on the table

The following statements cannot be executed if the table has pending definition changes:

- CREATE TRIGGER on the table
- CREATE TRIGGER of an INSTEAD OF trigger on a view that is dependent the table
- CREATE PERMISSION on the table or that references the table
- CREATE MASK on the table or that references the table
- CREATE FUNCTION of an inline SQL table function that references the table
- CREATE TABLE or ALTER TABLE that defines a materialized query table that references the table

The following statements cannot be executed if the table contains any columns with pending definition changes:

- CREATE VIEW that references a column with pending definition changes
- ALTER TABLE with the ADD VERSIONING clause that references a history table that contains columns with pending definition changes
- ALTER TABLE with the ENABLE ARCHIVE clause that references an archive table that contains columns with pending definition changes

Adding a LOB column:

If the table space that contains the table is implicitly created and you add a LOB column to the table, the following object are implicitly created:

- A LOB table space
- An auxiliary table
- An auxiliary index

If the base table is involved in a clone relationship, implicitly created LOB table spaces and implicitly created indexes are always created with the DEFINE YES attribute.

Adding a ROWID column:

When you add a ROWID column to an existing table, DB2 ensures that the same, unique row ID value is returned for a row whenever it is accessed. If the table already has an implicitly hidden ROWID column, DB2 also ensures that the values in the two ROWID columns are identical.

If the table space that contains the table is implicitly created and you add a ROWID column that is defined as GENERATED BY DEFAULT to the table, an enforcing index for the ROWID column is implicitly created. If the table already has an implicitly hidden ROWID column and the ROWID column that you add is defined as GENERATED BY DEFAULT, DB2 changes the implicitly hidden ROWID column to have the GENERATED BY DEFAULT attribute and does not implicitly create an enforcing index for the ROWID column.

When you add a ROWID column that is defined as GENERATED BY DEFAULT and the ROWID index is implicitly created, the privilege set requires the INDEX privilege on the table and the USE privilege on the buffer pool and the storage group. The implicitly created ROWID index is owned by the owner of the table.

Reorganizing a table space has no effect on the values in a ROWID column.

Adding an identity column:

When you add an identity column to a table that is not empty, DB2 places the table space that contains the table in the REORG pending state. When the REORG utility is subsequently run, DB2 generates the values for the identity column in all existing rows and then removes the REORG pending status. These values are guaranteed to be unique, and their order is system-determined.

Adding a row change timestamp column:

When you add a row change timestamp column to an existing table, the initial value for existing rows is not stored at the time of the ALTER statement. DB2 places the table space into an advisory-REORG pending state. When the REORG utility is subsequently run, DB2 generates the values for the row change timestamp column in all existing rows and then removes the REORG pending status. These values will not change unless the row is updated.

XML version support when adding an XML column:

When an XML column is added to a table that is in a universal table space, the XML column and the associated XML table will support XML versions if it is the first XML column in the table or if all the other XML columns in the table support XML versions. Similarly, when a clone table is associated

with the base table, any XML columns and associated XML tables will support XML versions if the existing XML columns in the base table support XML versions.

Effect of adding a column on views:
Adding a column to a table has no effect on existing views.

Considerations for implicitly hidden columns:
A column that is defined as implicitly hidden can be explicitly referenced on the ALTER statement. For example, an implicitly hidden column can be altered, can be specified as part of a referential constraint or a check constraint, or a materialized query table definition.

Cascaded effects of adding or altering a column:
Adding a column to a table has no cascaded effects to views that reference the table. For example, adding a column to a table does not cause the column to be added to any dependent views, even if those views were created with a SELECT clause. But altering a column can cause other cascaded effects. The following table lists the cascaded effect of altering the data type, precision, scale, or length of a column.

Table 106. Cascaded effect of altering a column's data type, precision, scale, or length

Operation	Effect
Alter of a column referenced by a view	If the data type, length, precision, or scale for a column is altered, all the views that are dependent on the altered table are reevaluated at alter time with the new column attributes. If errors are encountered during the view regeneration process, the ALTER TABLE statement fails. The new internal structure of each dependent view is not saved at alter time, and subsequent references to a dependent view will cause the view to be regenerated again. Use the ALTER VIEW statement to regenerate a dependent view and have the new internal structure saved.
Alter of a column referenced in the key of an index or a unique constraint (unique key or primary key)	The alter is allowed unless DECIMAL with a fraction is being converted to a floating value. In this case, the loss of precision can result in a loss of uniqueness. For numeric data type conversions, the index is placed in REBUILD-pending status. For character data type conversions, the index key columns are converted on first-write access. The index is not placed in REBUILD-pending status.
Alter of a column referenced in a package	The alter is allowed. All packages dependent on the table in which the column is being altered are invalidated.
Alter of a column referenced in the body of a user-defined function or procedure	Alter is allowed. If there is a package associated with the function or procedure, it is invalidated.
Alter of a column referenced in the parameter list of a user-defined function or procedure	Alter is allowed. The attributes of the existing function or procedure are unchanged. To access the new definition of the column, the function or procedure must be dropped and recreated.

Table 106. Cascaded effect of altering a column's data type, precision, scale, or length (continued)

Operation	Effect
Alter of a column referenced by a trigger	Alter is allowed. All trigger packages that are dependent on the table of the column are invalidated.
Alter of a column referenced in a CHECK constraint	Alter is not allowed.

Adding a partition:

When you add a partition to a table, if the boundary for the last partition was not previously enforced, it is enforced after the partition is added, and the last two logical partitions are left in REORG-pending (REORP) status. If the last partition before the new one is added was in REORG-pending status, the added partition is also placed in REORG-pending status.

Adding a partition for a table that is in a partition-by-growth table space and has LOB columns:

If a table resides in the partition-by-growth table space that has LOB columns, its associated LOB table spaces can be created either explicitly or implicitly when the base table is created, depending on value that is in effect for SQLRULES:

- When SQLRULES = STD, the LOB table space is created implicitly for the first partition or for the number of partitions in the NUMPARTS clause, if it is specified in the CREATE TABLESPACE statement.
- When SQLRULES = DB2, the table definition will remain incomplete until the LOB table space is explicitly created for the first partition or for the number of partitions in the NUMPARTS clause, if it is specified in the CREATE TABLESPACE statement. In this case, if the LOB table space is not created before the first SQL data change statement operates against the table, the table that resides in the partition-by-growth table space remains with its definition in incomplete state. The table cannot be updated through SQL or LOAD.

Attributes that are inherited from the previous LOB table space partition when a LOB table space is created implicitly:

The following attributes apply to implicitly created LOB table space:

- BUFFERPOOL
- DATASET
- ERASERULE
- GBPCACHE
- LOCKMAX
- LOG
- CLOSE
- DSSIZE
- LOCKSIZE

Row format for newly added partitions:

When the value of the RRF subsystem parameter is ENABLE, newly added partitions that are created using the ADD PARTITION clause (or partitions that are added because the table space is partition-by-growth) will be created in re-ordered row format. When the value of the RRF subsystem

parameter is DISABLE, newly added partitions will be created in basic row format, except for the following table spaces:

- For table spaces that are already using basic row format and that contain tables with edit procedures, newly created partition will always be in basic row format regardless of value of the RRF parameter.
- For table spaces that are already using re-ordered row format and that contain tables with edit procedures, newly created partition will always be in re-ordered row format regardless of value of the RRF parameter.
- Newly created partitions of an XML table space will always be in re-ordered format.

Rotating a partition from first to last:

Running ALTER TABLE to rotate the first logical partition to become the last logical partition can be very time consuming. During the reset operation, all rows from the partition are deleted. In addition, the keys for the deleted rows are also deleted from all nonpartitioned indexes, which requires that each nonpartitioned index must be scanned.

When you rotate partitions, if the boundary for the last partition was not previously enforced, it is enforced after ROTATE FIRST TO LAST is issued, and the last two logical partitions are left in REORG-pending (REORP) status. If the last partition before ROTATE FIRST TO LAST was issued was in REORG-pending status, the last two logical partitions are left in REORG-pending status.

Effect of changes on applications:

Applications might need to be changed to correspond to changes to the columns in a table. For example, if you increase the length of a column, you need to increase the length of variables into which that column is fetched. If you change the data type of a column, you also might need to change the data type of the corresponding variable to avoid performance degradation.

If you rename or drop a column, you need to change any references to that column to avoid unexpected results.

Invalidation of packages:

All of the packages that refer to the table are invalidated when any of the following conditions are true:

- The table is a created temporary table or a materialized query table.
- The table is changed to add or drop a materialized query definition.
- The AUDIT attribute of the table is changed.
- A DATE, TIME, TIMESTAMP WITHOUT TIME ZONE, or TIMESTAMP WITH TIME ZONE column is added and its default value for added rows is CURRENT DATE, CURRENT TIME, CURRENT TIMESTAMP (with corresponding timestamp precision) WITHOUT TIME ZONE, or CURRENT TIMESTAMP (with corresponding timestamp precision) WITH TIME ZONE respectively.
- A security label is added.
- The length attribute of a CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, BINARY, or VARBINARY column has changed. See Table 103 on page 1014.
- A column data type, precision, scale, or subtype is changed.
- A column is renamed.

- The table is partitioned and a partition is added or one of the existing partitions is changed or rotated
- An identity attribute of an identity column has changed
- A column is dropped

When a referential constraint is defined with a delete rule of CASCADE or SET NULL, all packages that refer to the parent table of the constraint are invalidated. Furthermore, all packages that refer to tables from which deletes cascade to this parent table are also invalidated.

Altering a base table or a user-maintained materialized query table to change it to a system-maintained materialized query table causes any packages that are dependent on the table to be invalidated because data change statements are not allowed on system-maintained materialized query tables. Altering a materialized query table to change it to a base table causes any packages that are dependent on the table to be invalidated because the REFRESH TABLE statement is invalid on a base table.

Invalidation of packages by RENAME COLUMN:

ALTER TABLE RENAME COLUMN will invalidate any package that is dependent on the table in which the column is renamed. Any attempt to execute the invalidated package will trigger an automatic rebind of the package.

The automatic rebind will fail if the column is referenced in the package because the referenced column no longer exists in the table. In this case, applications that reference the package need to be modified, recompiled, and rebound to return the expected result.

The automatic rebind will succeed in either of the following cases:

- The package does not reference the column. In this case, the renaming of the column does not affect the query results that are returned by the package. The application does not need to be modified as a result of renaming the column.
- The package does reference the column, but after the column is renamed, another column with the name of the original column is added to the table. In this case, any query that references the name of the original column might return a different result set. In order to restore the expected results, the application would need to be modified to specify the new column name.

Example: The following scenario shows how renaming a column can cause a package to return unexpected results:

```
CREATE TABLE MYTABLE (MYCOL1 INT);
INSERT INTO TABLE MYTABLE
 VALUES (1);
SELECT MYCOL1 FROM MYTABLE -- this is the statement in
               --  the package MYPACKAGE,
                           --  the query returns
                           --  a value of 1
ALTER TABLE MYTABLE
   RENAME COLUMN
    MYCOL1 TO MYCOL2;      -- MYPACKAGE is invalidated
                           --  and automatic rebind
                           --  of MYPACKAGE will fail
                           --  at this point
ALTER TABLE MYTABLE
   ADD COLUMN MYCOL1 VARCHAR(10);  -- automatic rebind
```

```
                                        --  of MYPACKAGE
                                        --  will be successful
INSERT INTO TABLE MYTABLE (MYCOL1)
   VALUES ('ABCD');
```

At this point an application executes MYPACKAGE, which results in a successful automatic rebind. However, the statement in the package will return 'ABCD' instead of the expected '1'.

Dropping constraints and check pending status:
If a table space or partition is in check pending status because it contains a table with rows that violate constraints, dropping the constraints removes the check pending status.

Altering materialized query tables:
The ALTER TABLE statement can be used to register an existing table at the current server as a materialized query table, change the attributes of an existing materialized query table, or change an existing materialized query table into a base table.

The isolation level at the time when a base table is first altered to become a materialized query table by the ALTER TABLE statement is the isolation level for the materialized query table.

Altering a table to change it to a materialized query table with query optimization enabled makes the table eligible for use in query rewrite immediately. Therefore, pay attention to the accuracy of the data in the table. If necessary, the table should be altered to a materialized query table with query optimization disabled, and then the table should be refreshed and enabled with query optimization.

When a base table is altered into a materialized query table or a user-maintained query table is altered into a system-maintained one, the REFRESH_TIME column of the row for the table in SYSIBM.SYSVIEWS contains the current timestamp. When a system-maintained materialized query table is altered into a user-maintained materialized query table, the REFRESH_TIME column of the row for the table in SYSIBM.SYSVIEWS does not change.

The LOAD utility is not allowed on a system-maintained query table, but it is allowed on a user-maintained materialized query table.

Considerations for running utilities while altering tables:
You cannot execute the ALTER TABLE statement while a utility has control of the table space that contains the table.

Restrictions on field procedures, edit procedures, and validation exit procedures:
Field procedures, edit procedures that are defined as WITH ROW ATTRIBUTES, and validation exit procedures cannot be used on tables that have column names that are larger than 18 EBCDIC bytes. If you have tables that have field procedures or validation exit procedures and you add a column where the column name is larger than 18 bytes, the field procedures and validation exit procedures for the table will be invalidated.

Consider using triggers to replace the functionality on field procedures, edit procedures that are defined as WITH ROW ATTRIBUTES, and validation exit procedures on tables where the column names are larger than 18 EBCDIC bytes.

Restrictions on SQL data change statements in the same commit scope as ALTER TABLE:

SQL data change statements that affect an index cannot be performed in the same commit scope as ALTER TABLE statements that affect that index.

Restrictions on DATA CAPTURE CHANGES:

If the table is in advisory REORG-pending state, you cannot alter the table to use the DATA CAPTURE CHANGES clause.

Capturing changes to the DB2 catalog:

To have logged changes to a DB2 catalog table augmented with information for data capture, specify ALTER TABLE *xxx* DATA CAPTURE CHANGES where *xxx* is the name of a catalog table (SYSIBM.*xxx*). Data capture of catalog table changes provides the possibility of creating and managing a shadow of the catalog.

When changes to the hash organization of a table take place:

An alter of the table that uses hash organization will take effect immediately in terms of enforcing the unique hash key. However, the physical organization of the table space is converted to hash organization after REORG.

In a range-partitioned universal table space, if individual partitions are altered to specify HASH SPACE, the new hash space values take effect after the REORG utility is run on the individual partitions.

Buffer pool, DSSIZE, and MAXPARTITIONS considerations for tables using hash organization:

DB2 will calculate an optimum buffer pool size for hash organization based on the definition of the table and validate the calculated buffer pool size with the buffer pool of the explicitly created table space. If the buffer pool sizes are different, DB2 will return an error.

If the table is a range-partitioned universal table space, the DSSIZE value for the table space must be large enough to fit the HASH SPACE specification for each partition.

If the table is in a partition-by-growth table space, the total space calculated from the DSSIZE and MAXPARTITIONS values for the table space must be large enough for the implicitly or explicitly specified HASH SPACE.

Changing the hash space value:

To change the HASH SPACE value for all partitions of a range-partitioned universal table space or to change the total HASH SPACE for a partition-by-growth table space, use the ALTER ORGANIZATION SET HASH SPACE (*integer*) clause. To change HASH SPACE value for more than one, but not all partitions of a range-partitioned universal table space you must specify separate ALTER TABLE statements for each partition and specify the ALTER PARTITION (*integer*) and HASH SPACE (*integer*) clauses.

Hash space and DB2 page size:

If the specified hash space is less than or equal to 64 MB (the DB2 default), DB2 will add extra space for DB2 system pages. If the specified hash space is greater than 64 MB, DB2 will use part of the hash space for DB2 system pages. The amount of space needed for DB2 system pages depends on SEGSIZE and PAGESIZE. The larger the SEGSIZE and/or PAGESIZE

becomes, the larger the requirement for DB2 system pages. DB2 can reserve up to 5 MB for system pages for the highest SEGSIZE value (64) and PAGESIZE value (32K).

Hash space and DSSIZE:
Depending on certain table space characteristics, DB2 needs to reserve space for the hash overflow area. Therefore, the amount of hash space cannot be equal to the DSSIZE value. The maximum amount of hash space that can be specified is approximately 20% less than the DSSIZE value. DB2 returns an error if the amount of hash space is too large. If the amount of hash space is too large, specify a larger value of DSSIZE, or decrease the amount of hash space.

Specifying APPEND for tables that use hash organization:
Append processing is not applicable to tables with hash organization since there is no key clustering in hash organization. For insert operations into tables with hash organization, DB2 will use the internal hash algorithm to determine the location of the row.

Restrictions for tables with hash organization:
Tables that use hash organization are subject to the following restrictions:

- If the table already uses hash organization, DB2 will returns an error.
- A table that is defined to use hash organization cannot be created in a LOB table space or XML table space.
- The data type of columns that are specified in a hash key cannot be changed.
- Partition level REORG is not allowed after the table is changed using the ALTER ADD HASH ORGANIZATION clause or the ALTER DROP ORGANIZATION clause.
- The MAXROWS clause is applicable only to the hash overflow area of the table space for tables with hash organization. The fixed hash area of each page will contain as many rows as it can hold, up to a maximum of 255.
- DB2 implicitly creates a hash overflow index when hash organization is added to a table. The hash overflow index is in rebuild-pending state until the REORG utility is run.

Row access control that is activated explicitly:
The ACTIVATE ROW ACCESS CONTROL clause is used to activate row access control for a table. When this happens, a default row permission is implicitly created and allows no access to any rows of the table, unless later another enabled row permission exists that provides access for the authorization IDs or roles that are specified in the definition of the permission. The default row permission is always enabled.

When the table is referenced in a data manipulation statement, all enabled row permissions that have been created for the table, including the default row permission, are implicitly applied by DB2 to control which rows in the table are accessible. A row access control search condition is derived by application of the logical OR operator to the search condition in each enabled row permission. This derived search condition acts as a filter to the table before any user specified operations such as predicates, grouping, ordering, etc. are processed. This derived search condition permits the authorization IDs or roles that are specified in the permission definitions to access certain rows in the table. See the description of subselect for information on how the application of enabled row permissions affects the

fetch operation. See the data change statements for information on how the application of enabled row permissions affects the data change operation.

When the ACTIVATE ROW ACCESS CONTROL clause is used, all the packages and dynamic cached statements that reference the table are invalidated.

Row access control remains enforced until the DEACTIVATE ROW ACCESS CONTROL clause is used to stop enforcing it.

Implicit object that is created when row access control is activated for a table:
When the ACTIVATE ROW ACCESS CONTROL clause is used to activate row access control for a table, DB2 implicitly creates a default row permission for the table. The default row permission prevents all access to the table. The implicitly created row permission is in the same schema of the base table and has a name in the form of SYS_DEFAULT_ROW_PERMISSION__*table-name* ... up to 128 UTF-8 bytes. Notice two underscores after "PERMISSION". If this name is not unique, the last 4 bytes are reserved for a unique number '*nnnn*', where '*nnnn*' is a four alphanumeric characters starting at '0000' and is incremented by 1 value each time until a unique name is found. The owner of the default row permission is SYSIBM.

The default row permission is always enabled.

The default row permission is dropped when row access control is deactivated or when the table is dropped.

Activating column access control:
The ACTIVATE COLUMN ACCESS CONTROL clause is used to activate column access control for a table. The access to the table is not restricted but when the table is referenced in a data manipulation statement, all enabled column masks that have been created for the table are applied to mask the column values referenced in the final result table of the queries or to determine the new values used in the data change statements.

When column masks are used to mask the column values, they determine the values in the final result table. If a column has a column mask and the column (a simple reference to a column name or embedded in an expression) appears in the outermost select list, the column mask is applied to the column to produce the values for the final result table. If the column does not appear in the outermost select list but it participates in the final result table, for example, it appears in a materialized table expression or view, the column mask is applied to the column in such a way that the masked value is included in the result table of the materialized table expression or view so that it can be used in the final result table.

The application of column masks does not interfere with the operations of other clauses within the statement such as the WHERE, GROUP BY, HAVING, SELECT DISTINCT, and ORDER BY. The rows returned in the final result table remain the same, except that the values in the resultant rows might have been masked by the column masks. As such, if the masked column also appears in an ORDER BY sort-key, the order is based on the original column values and the masked values in the final result table might not reflect that order; similarly, the masked values might not reflect the uniqueness enforced by SELECT DISTINCT. If the masked column is embedded in an expression, the result of the expression can become different because the column mask is applied on the column before the expression evaluation can take place. If the expression in a query is the

same as the expression used to mask the column value in the column mask definition, the result of the expression in the query might remain unchanged. For example, the expression in the query is 'XXX-XX-' || SUBSTR(SSN, 8, 4) and the same expression appears in the column mask definition. In this particular example, the user can replace the expression in the query with column SSN to avoid the same expression gets evaluated twice.

The following are the contexts where the column masks are used by DB2 to mask the column values for the result of a query. Certain restrictions might apply to some contexts. Those restrictions are described in a separate list.

- the outermost SELECT clause of a SELECT or SELECT INTO statement, or if the column does not appear in the outermost select list but it participates in the final result table, the outermost SELECT clause of the corresponding materialized table expression or view where the column appears.
- the outermost SELECT clause of a SELECT FROM INSERT, UPDATE, DELETE, or MERGE statement
- the outermost SELECT clause that are used to derive the new values for an INSERT, UPDATE, or MERGE statement, or a SET *transition-variable* assignment statement
- the same applies to a scalar-fullselect expression that does not use set operators and appears in the outermost SELECT clause of the above statements, the right side of a SET variable assignment statement, the VALUES INTO statement, or the VALUES statement.
- the same applies to the SQL statements or the equivalences such as the assignment statement that appears in a native SQL procedure or a non-inline user-defined SQL function.

If a CASE expression appears in the above contexts, the column masks are not applied in the search conditions of the WHEN clauses.

A column mask is created as a stand alone object without knowing all of the contexts in which it might be used. To mask a column value in the final result table, the column mask definition is merged into the statement by DB2. When the column mask definition is brought into the context of the statement, it might conflict with certain SQL semantics in the statement. Therefore, in some situations, the combination of the statement and the application of a column mask can return an error. The following describes when the error might be returned:

1. The column masks cannot be applied to the columns in the select lists that derive the final result table of set operations because one of the set operators that are used to derive the final result table is EXCEPT ALL, EXCEPT DISTINCT, INTERSECT ALL, or INTERSECT DISTINCT.
2. The column mask cannot be applied to the column in the select lists of a scalar-fullselect expression if the result of scalar-fullselect expression is derived from set operation EXCEPT or INTERSECT.
3. If the subselect contains a GROUP BY clause, the column mask cannot be applied to a column in the corresponding select list if none of the following conditions is satisfied:

- The column must identify a *column-name* in the GROUP BY clause and the column must not be referenced in an expression in the GROUP BY clause. Furthermore, its column mask definition must satisfy the following condition:
  - any columns that are referenced in the column mask definition that come from the same table of the column to which the column mask is applied must identify a column-name in the GROUP BY clause
  - the column mask must not be referenced in an expression in the GROUP BY clause
- The column must be specified under an aggregate function and its column mask definition must satisfy the following conditions:
  - The column mask definition must not reference a scalar-fullselect
  - The column mask definition must not reference an aggregate function

4. If the subselect contains a GROUP BY clause, and a column in the corresponding select list maps directly or indirectly to a column name or an expression in a materialized table expression or view, the column in the subselect where the GROUP BY is specified must be specified under an aggregate function.
5. If the subselect does not contain a GROUP BY clause, and a column in the corresponding select list is specified under an aggregate function, the column mask cannot be applied if the column mask definition references:
   - a scalar-fullselect
   - an aggregate function
6. If the FROM clause in a subselect references a recursive common table expression, and if the result of the recursive common table expression is used to derive the final result table, the column mask cannot be applied to a column that is referenced in the fullselect of the recursive common table expression.
7. If the FROM clause in a subselect contains a *data-change-table-reference*, and if an INCLUDE clause is specified as part of the SQL data change statement, the column mask cannot be applied to the columns that are used to derive the values for these additional columns in the outermost select list.
8. If the FROM clause in a subselect references an external table user-defined function or an inline SQL table user-defined function, and if the result of the function is used to derive the final result table, the column mask cannot be applied to the column that is an argument of the function.
9. If an OLAP specification is referenced in a select list that derives the final result table, the column mask cannot be applied to the column that is referenced in the partitioning expression or the sort key expression of the OLAP specification.
10. If a user-defined function is defined with the NOT SECURED option, the argument of the function must not reference a column for which a column mask is enabled and the column access control is activated for its table. This rule applies to user-defined functions that are referenced anywhere in the statement.

To avoid the above error situations at bind time, one of the following actions must be taken:

- modify or remove the above contexts from the statement
- disable the column mask
- drop the column mask, modify the definition, and recreate the column mask
- deactivate the column access control for the table

In other situations, if the statement contains a SELECT DISTINCT, and a column mask is applied to a column that directly or indirectly derives the result of SELECT DISTINCT, the statement might return a result that is not deterministic. The following examples illustrate when such results might be returned:

1. If the column mask definition references other columns from the same table of the column to which the column mask is applied, the result of SELECT DISTINCT can not be deterministic.
2. If the column is referenced in the argument of built-in scalar functions (such as COALESCE, IFNULL, NULLIF, MAX, MIN, LOCATE, TOTALORDER), the result of SELECT DISTINCT might not be deterministic.
3. If the column is referenced in the argument of an aggregation function, the result of SELECT DISTINCT might not be deterministic. If DISTINCT is specified, the argument of the function must not reference a column with a column mask.
4. If the column is embedded in an expression and the expression contains a function that is not deterministic or has an external action, the result of SELECT DISTINCT might not be deterministic.

With UNION DISTINCT, the elimination of the duplicate rows is based on the unmasked values in R1 and R2. Because all rows are from R1 or R2, the output values in the result table of the union may vary when one or more of the following conditions occur:

- The expression corresponding to the *n*th column in R1 references columns with column masks, but the expression corresponding to the *n*th column in R2 does not, or vise versa.
- The expressions corresponding to the *n*th column in R1 and R2 reference columns with different column masks.
- The column mask definition references columns that are not the same target column for which the column mask is defined, and those columns are not part of the UNION DISTINCT operation. It is recommended that the column mask definition does not reference other columns from the target table.

For example, a row in R1 is derived from the masked value, and a row in R2 is derived from the unmasked value. If the row in the result table is from R1, the masked value is returned. If the row in the result table is from R2, the unmasked value is returned.

EXCEPT and INTERSECT can be intermixed with UNION if the rows in R1 and R2 for EXCEPT and INTERSECT do not reference columns with column masks

If the column is not nullable, most likely its column mask definition will not consider a null value for the column. After the column access control is activated for the target table, if the target table is the null-padded table in an outer join operation, the column value in the final result table might be a null.

When the columns are used to derive the new values for an INSERT, UPDATE, MERGE, or a SET transition-variable assignment statement, the original column values, not the masked values, are used. If the columns have column masks, those column masks are applied to ensure the evaluation of the access control rules at run time masks the column to itself, not to a constant or an expression. This is to ensure the masked values are the same as the original column values. If a column mask does not mask the column to itself, the existing row is not updated or the new row is not inserted and an error is returned at run time. The rules that are used to apply column masks in order to derive the new values follow the same rules described above for the final result table of a query. See the data change statements for how the column masks are used to affect the insertability and updatability

A column mask can be applied only to a base table column. If a materialized table expression, materialized view, or common table expression column is involved in the final result table, the above error situations can occur inside the materialized table expression, materialized view, or common table expression definition.

Column access control does not affect the XMLTABLE built-in function. If the input to the XMLTABLE function is a column with a column mask, the column mask is not applied.

When the ACTIVATE COLUMN ACCESS CONTROL clause is used, all the packages and dynamic cached statements that reference the table are invalidated. However, if no enabled column mask exists for the table, the invalidation does not occur.

Column access control remains activated until the DEACTIVATE COLUMN ACCESS CONTROL clause is used to stop enforcing it.

Row and column access control are not enforced when EXPLAIN tables are populated by DB2:

Row and column access control can be enforced for EXPLAIN tables. However, the enabled row permissions and column masks are not applied when DB2 inserts rows into those tables.

Stop enforcing row or column access control:

The DEACTIVATE ROW ACCESS CONTROL clause is used to stop enforcing row access control for a table. The default row permission is dropped. Thereafter, when the table is referenced in a data manipulation statement, explicitly created row permissions are not applied. The table is accessible based on the granted privileges.

The DEACTIVATE COLUMN ACCESS CONTROL clause is used to stop enforcing column access control for a table. Thereafter, when the table is referenced in a data manipulation statement, the column masks are not applied. The unmasked column values are used for the final result table.

The explicitly created row permissions or column masks, if any, remain but have no effect.

All the packages and dynamic cached statements that reference the table are invalidated when row or column access control is deactivated.

Secure triggers for row and column access control:

Triggers are used for database integrity, and as such a balance between row and column access control (security) and database integrity is needed. Enabled row permissions and column masks are not applied to the initial values of transition variables and transition tables. Row and column access

control enforced for the triggering table is also ignored for any transition variables or transition tables referenced in the trigger body. To ensure there is no security concern for SQL statements in the trigger action to access sensitive data in transition variables and transition tables, the trigger must be created or altered with the SECURED option. If a trigger is not secure, row and column access control cannot be enforced for the triggering table.

Secure user-defined functions for row and column access control:
If a row permission or column mask definition references a user-defined function, the function must be altered with the SECURED option because the sensitive data might be passed as arguments to the function.

DB2 considers the SECURED option an assertion that declares the user has established a change control audit procedure for all changes to the user-defined function. It is assumed that such a control audit procedure is in place for all versions of the user-defined function, and that all subsequent ALTER FUNCTION statements or changes to external packages are being reviewed by this audit process.

Database operations where row and column access control is not applicable:
Row and column access control must not compromise database integrity. Columns involved in primary keys, unique keys, indexes, check constraints, and referential integrity (RI) must not be subject to row and column access control. Column masks can be defined for those columns but they are not applied during the process of key building or constraint or RI enforcement.

Read-only cursors and read-only views:
The rules that are used to determine a read-only cursor or a read-only view remain unaffected by row and column access control because those rules are determined at bind time. The effect of application of enabled column masks is not known until run time. Therefore, the data change operation on a writable cursor or a writable view could still fail at run time.

Considerations for adding a column to a system-period temporal table or archive-enabled table:

- If the data type of the column is a distinct type:
  - The owner of the history table or archive table must implicitly or explicitly have the USAGE privilege on the distinct type.
  - If the distinct type is unqualified, its schema matches the schema for the following objects:
    - The implicit schema for the distinct type for the column in the history table is the same as the implicit schema that is determined for the distinct type in the system-period temporal table.
    - The implicit schema for the distinct type for the column in the archive table is the same as the implicit schema that is determined for the distinct type in the archive-enabled table.
  -
- The syntax LONG VARCHAR or LONG VARGRAPHIC must not be specified when you add a column to these types of tables. Use VARCHAR or VARGRAPHIC instead.
- If the data type of the column is a LOB and the INLINE LENGTH clause is not specified, DB2 determines the length. The implicit inline length

that is used for the column in the system-period temporal table or archive-enabled table is also used for the corresponding column in the history table or archive table.

- If the data type of the column is a LOB, auxiliary objects are implicitly created for it in the system-period temporal table or archive-enabled table. Auxiliary objects are also created for the corresponding column of the history table or archive table.

Alternative syntax and synonyms:

To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following clauses:

- NOCACHE (single clause) as a synonym for NO CACHE
- NOCYCLE (single clause) as a synonym for NO CYCLE
- NOMINVALUE (single clause) as a synonym for NO MINVALUE
- NOMAXVALUE (single clause) as a synonym for NO MAXVALUE
- NOORDER (single clause) as a synonym for NO ORDER
- PART *integer* VALUES can be specified as an alternative to PARTITION *integer* ENDING AT.
- VALUES as a synonym for ENDING AT
- DEFINITION ONLY as a synonym for WITH NO DATA
- SET MATERIALIZED QUERY AS DEFINITION ONLY as a synonym for DROP MATERIALIZED QUERY
- SET SUMMARY AS DEFINITION ONLY as a synonym for DROP MATERIALIZED QUERY
- SET MATERIALIZED QUERY AS (fullselect) as a synonym for ADD MATERIALIZED QUERY (fullselect)
- SET SUMMARY AS (fullselect) as a synonym for ADD MATERIALIZED QUERY (fullselect)
- TIMEZONE can be specified as an alternative to TIME ZONE.

Examples

Example 1: Column DEPTNAME in table DSN8B10.DEPT was created as a VARCHAR(36). Increase its length to 50 bytes. Also, add the column BLDG to the table DSN8B10.DEPT. Describe the new column as a character string column that holds SBCS data.

```
ALTER TABLE DSN8B10.DEPT
  ALTER COLUMN DEPTNAME SET DATA TYPE VARCHAR(50)
  ADD BLDG CHAR(3) FOR SBCS DATA;
```

Example 2: Assign a validation procedure named DSN8EAEM to the table DSN8B10.EMP.

```
ALTER TABLE DSN8B10.EMP
  VALIDPROC DSN8EAEM;
```

Example 3: Disassociate the current validation procedure from the table DSN8B10.EMP. After the statement is executed, the table no longer has a validation procedure.

```
ALTER TABLE DSN8B10.EMP
  VALIDPROC NULL;
```

Example 4: Define ADMRDEPT as the foreign key of a self-referencing constraint on DSN8B10.DEPT.

```
ALTER TABLE DSN8B10.DEPT
  FOREIGN KEY(ADMRDEPT) REFERENCES DSN8B10.DEPT ON DELETE CASCADE;
```

Example 5: Add a check constraint to the table DSN8B10.EMP which checks that the minimum salary an employee can have is $10,000.

```
ALTER TABLE DSN8B10.EMP
  ADD CHECK (SALARY >= 10000);
```

Example 6: Alter the PRODINFO table to define a foreign key that references a non-primary unique key in the product version table (PRODVER_1). The columns of the unique key are VERNAME, RELNO.

```
ALTER TABLE PRODINFO
  FOREIGN KEY (PRODNAME,PRODVERNO)
    REFERENCES PRODVER_1 (VERNAME,RELNO) ON DELETE RESTRICT;
```

Example 7: Assume that table DEPT has a unique index defined on column DEPTNAME. Add a unique key constraint named KEY_DEPTNAME consisting of column DEPTNAME to the DEPT table:

```
ALTER TABLE DSN8B10.DEPT
  ADD CONSTRAINT KEY_DEPTNAME UNIQUE( DEPTNAME );
```

Example 8: Register the base table TRANSCOUNT as a materialized query table. The result of the fullselect must provide a set of columns that match the columns in the existing table (same number of columns, same column definitions, and same names). So that you can maintain the table with insert, update, and delete operations as well as the REFRESH TABLE statement, define the materialized query table as user-maintained.

```
ALTER TABLE TRANSCOUNT ADD MATERIALIZED QUERY
  (SELECT ACCTID, LOCID, YEAR, COUNT(*) as cnt
   FROM TRANS
   GROUP BY ACCTID, LOCID, YEAR )
   DATA INITIALLY DEFERRED
   REFRESH DEFERRED
   MAINTAINED BY USER;
```

Example 9: Assume that table TB1 has a column, COL1 that is defined as CHAR(4) FOR BIT DATA WITH DEFAULT 'AB'. The value that is stored in the table will be X'C1C24040'. After the following ALTER TABLE statement is run, the resulting value that is stored in the table will be BX'C1C240400000':

```
 ALTER TABLE TB1
ALTER COLUMN COL1
 SET DATA TYPE BINARY(6);
```

Examples for column access controls

Example 1:

Based on the data in the CUSTOMER table, the SELECT DISTINCT statement returns one row with the SALARY value 100,000. A column mask, SALARY_MASK, is created to mask the salary value. After column access control is activated for the CUSTOMER table, the column mask is applied to SALARY column. A user with the 'MGR' ID (or role) issues a SELECT DISTINCT statement. The SELECT DISTINCT statement still returns one row because the removal of duplicates is based on the unmasked value of the SALARY column, but the value that is returned in that row is based on the masked SALARY value, which can be either 125,000 or 110,000.

The table CUSTOMER contains:

SALARY	COMMISSION	EMPID
100,000	25,000	123456
100,000	10,000	654321

```
CREATE MASK SALARY_MASK ON CUSTOMER
 FOR COLUMN SALARY RETURN
    CASE WHEN(SESSION_USER = 'MGR')
        THEN SALARY + COMMISSION
             ELSE SALARY
         END
  ENABLE;

COMMIT;

ALTER TABLE CUSTOMER
 ACTIVATE COLUMN ACCESS CONTROL;

COMMIT;

SELECT DISTINCT SALARY FROM CUSTOMER;
```

Example 2:

Based on the data in T1 abd T2 tables, the SELECT DISTINCT statement using the COALESCE function returns one row with the T1.C1 value of 1. A column mask, C1_MASK, is created to mask the value of T1.C1. After column access control is activated for table T1, the column mask is applied to column C1 of table T1. A user with the 'EMP' ID (or role) issues a SELECT DISTINCT statement. The SELECT DISTINCT statement still returns one row because the removal of duplicates is based on the unmasked value of T1.C1 from the COALESCE function, but the value that is returned in that row is based on the masked value of T1.C1 from the COALESCE function. The returned value can be either 2 or 3.

```
INSERT INTO T1(C1) VALUES(1);
INSERT INTO T1(C1) VALUES(1);
INSERT INTO T2(C1) VALUES(2);
INSERT INTO T2(C1) VALUES(3);

CREATE MASK C1_MASK ON T1
 FOR COLUMN C1 RETURN
   CASE WHEN(SESSION_USER = 'EMP')
       THEN NULL
           ELSE C1
       END
  ENABLE;

COMMIT;

ALTER TABLE T1
   ACTIVATE COLUMN ACCESS CONTROL;

COMMIT;

SELECT DISTINCT COALESCE(T1.C1, T2.C1) FROM T1, T2;
```

Example 3:

Based on the data in the CUSTOMER table, the maximum income is the same in the states CA and IL, 50,000, thus, the SELECT DISTINCT statement returns one row. A column mask, INCOME_MASK, is created to mask the income value. After column access control is activated for the CUSTOMER table, the column mask is applied to the INCOME column before the MAX aggregate function is evaluated. However, the INCOME_ MASK column mask, masks the income value of 0 as 100,000 in state IL.

As a result, the maximum income becomes 100,000 for state IL, but the maximum income is still 50,000 for state CA. X.B is used in a predicate in the SELECT DISTINCT statement, therefore, the original INCOME values and the original results of the MAX(INCOME) function must be preserved. So the SELECT DISTINCT statement still returns one row, but the value in that row might not be deterministic, that is, the value might be 50,000 from the 'CA' row or might be 100,000 from the 'IL' row.

The CUSTOMER table contains:

STATE	INCOME
CA	40,000
CA	50,000
IL	0
IL	10,000
IL	50,000

```
CREATE MASK INCOME_MASK ON CUSTOMER
 FOR COLUMN INCOME RETURN
     CASE WHEN(INCOME = 0)
        THEN 100000
            ELSE INCOME
     END
 ENABLE;

COMMIT;

ALTER TABLE CUSTOMER
  ACTIVATE COLUMN ACCESS CONTROL;

COMMIT;

SELECT DISTINCT B FROM
 (SELECT STATE, MAX(INCOME) FROM CUSTOMER
  GROUP BY STATE)
  X(A, B)
    WHERE B > 10000;
```

Example 4:

The expression `INCOME + RAND()` is not deterministic because the RAND function is not deterministic. Based on the data in the CUSTOMER table, the SELECT DISTINCT statement will, most likely, return two distinct rows. However, it could return only one row. A column mask, INCOME_MASK, is created to mask the income value. After column access control is activated for the CUSTOMER table, the column mask is applied to the INCOME column, which causes the masked value for both rows to be the same. Because the RAND function is not deterministic, the SELECT DISTINCT statement will, most likely, still return two distinct rows, but it could return only one row.The uncertainty caused by the RAND function causes the result of the SELECT DISTINCT statement to not be deterministic.

The CUSTOMER table contains:

STATE	INCOME
CA	40,000
CA	50,000

```
CREATE MASK INCOME_MASK ON CUSTOMER
 FOR COLUMN INCOME RETURN
   CASE WHEN(INCOME = 40,000)
      THEN 50000
           ELSE INCOME
       END
 ENABLE;

COMMIT;

ALTER TABLE CUSTOMER
  ACTIVATE COLUMN ACCESS CONTROL;

COMMIT;

SELECT DISTINCT A FROM
  (SELECT INCOME + RAND() FROM CUSTOMER)
 X(A)
  WHERE A > 10000;
```

Example 5:

A column mask, STATE_MASK, is created for the STATE column of the CUSTOMER table to return a value that shows the city name with the state if the city is SJ, SFO, or OKLD. Otherwise the city is not returned, just he state. After column access control is activated for the CUSTOMER table, a SELECT statement which groups results using the STATE column is issued. However, because the CITY column that is referenced in the STATE_MASK column mask is not a grouping column, a bind time error is returned to signify that the STATE_MASK column mask is not appropriate for this statement.

The CUSTOMER table contains:

STATE	**CITY**	**INCOME**
CA	SJ	40,000
CA	SC	30,000
CA	SB	60,000
CA	SFO	80,000
CA	OKLD	50,000
CA	SJ	70,000
NY	NY	50,000

```
CREATE MASK STATE_MASK ON CUSTOMER
 FOR COLUMN STATE RETURN
    CASE WHEN(CITY = 'SJ')
         THEN CITY||', '||STATE
       WHEN(CITY = 'SFO')
       THEN CITY||', '||STATE
       WHEN(CITY = 'OKLD')
       THEN CITY||', '||STATE
       ELSE ' , '||STATE
        END
 ENABLE;

COMMIT;

ALTER TABLE CUSTOMER
 ACTIVATE COLUMN ACCESS CONTROL;

COMMIT;
```

```
SELECT STATE, AVG(INCOME) FROM CUSTOMER
 GROUP BY STATE
  HAVING STATE = 'CA';
```

ALTER TABLESPACE

The ALTER TABLESPACE statement changes the description of a table space at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

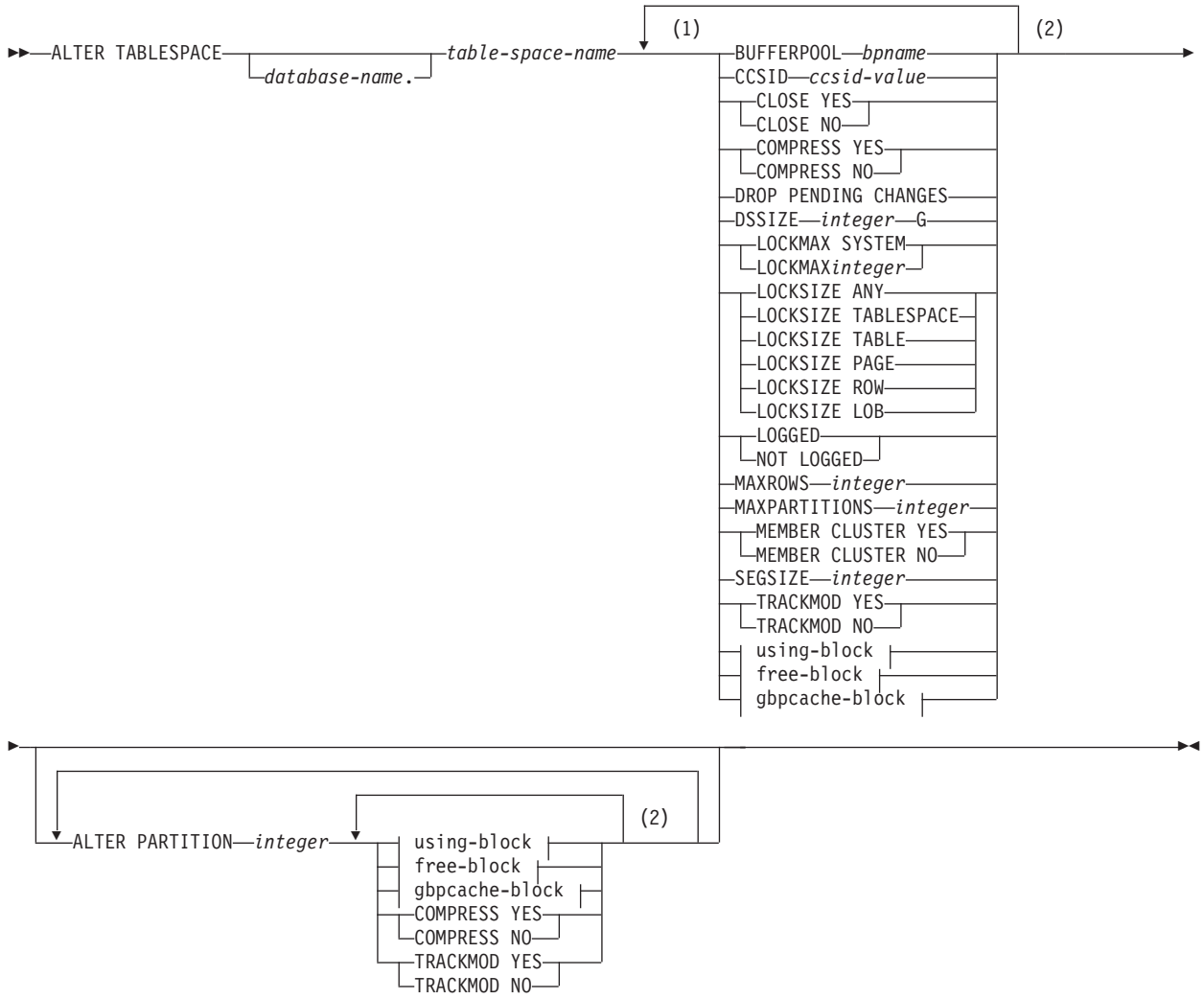
The privilege set that is defined below must include at least one of the following:
- Ownership of the table space
- DBADM authority for its database
- SYSADM or SYSCTRL authority
- System DBADM

If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

If BUFFERPOOL or USING STOGROUP is specified, additional privileges might be required, as explained in the description of those clauses.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the package. If the statement is dynamically prepared, the privilege set is the union of the privilege sets that are held by each authorization ID and role of the process.

Syntax

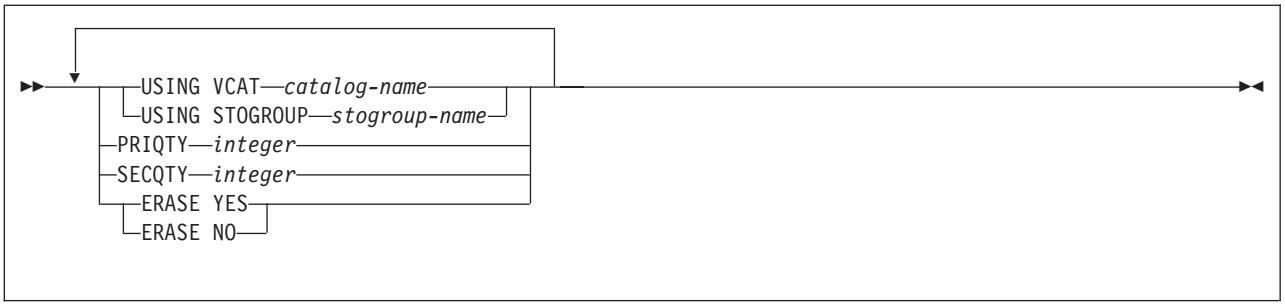


Notes:

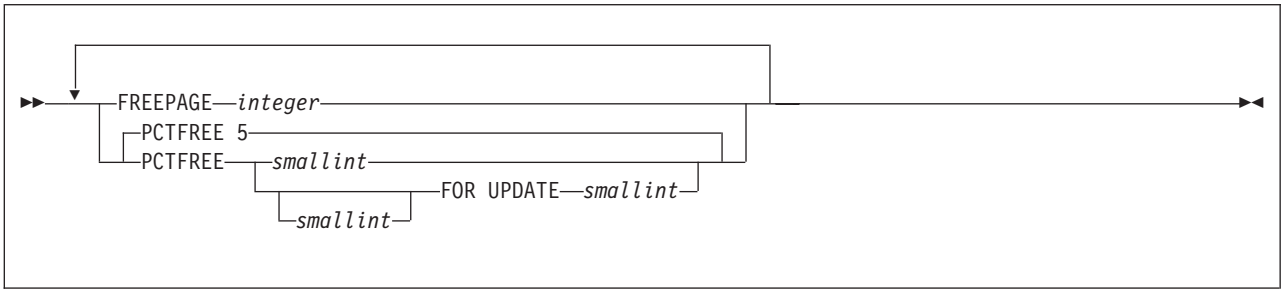
1 If you specify DROP PENDING CHANGES, DSSIZE, or SEGSIZE, no other clauses can be specified in the same ALTER TABLESPACE statement.

2 The same clause must not be specified more than one time in a single ALTER TABLESPACE statement. For example, if TRACKMOD YES is specified at the table space level, it must not be specified after ALTER PARTITION.

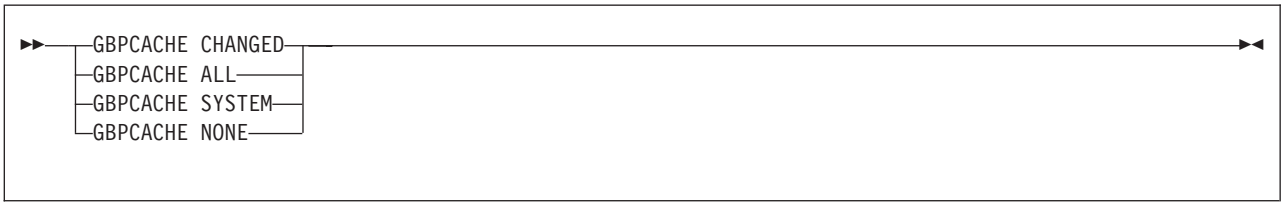
using-block:



free-block:



gbpcache-block:



Description

database-name.table-space-name
Identifies the table space that is to be altered. The name must identify a table space that exists at the current server. Omission of *database-name* is an implicit specification of DSNDB04.

If you identify a partitioned table space, you can use the PARTITION clause.

BUFFERPOOL *bpname*
Identifies the buffer pool that is to be used for the table space. *bpname* must identify an activated buffer pool.

The privilege set must include SYSADM or SYSCTRL authority or the USE privilege for the buffer pool.

If *bpname* specifies a buffer pool with a smaller page size than the current page size, the maximum record size of all tables in the table space must fit in the smaller page size.

If *bpname* specifies a buffer pool with a different page size, the table space can only be a universal table space (excluding XML table spaces) or a LOB table space. If the table space is a partition-by-growth universal table space, the page size must be valid depending on the values that are in effect for the MAXPARTITIONS and DSSIZE options of the table space. If the table space is a range-partitioned universal table space, the page size must be valid

depending on the values that are in effect for the current number of partitions and the DSSIZE option of the table space.

The buffer pool change is a pending change to the definition of the table space if the data sets of the table space are already created and if one of the following conditions is true:

- Pending definition changes already exist for the table space or any objects within the base table space.
- The specified buffer pool has a different page size than the buffer pool that is currently being used for the table space.

Otherwise, the change is considered an immediate change.

If the change is considered an immediate change, the change to the description of the table space takes effect the next time the data sets of the table space are opened. The data sets can be closed and reopened by using a STOP DATABASE command to stop the table space followed by a START DATABASE command to start the table space.

If the change is a pending change to the definition of the table space, the changes are not reflected in the definition or data at the time the ALTER TABLESPACE statement is issued. Instead, the entire table space is placed in an advisory REORG-pending state (AREOR). A subsequent reorganization of the entire table space will apply the pending definition changes to the definition and data of the table space.

When the pending page size change is applied, if the table space is a universal table space that uses partition-by-growth organization, the number of partitions is determined based on the amount of existing data and the new page size value. Changing the page size to be larger can cause automatic creation of additional partitions. If LOB columns exist, additional LOB table spaces and auxiliary objects are implicitly created for the newly-created partitions independent of whether SQLRULES (DB2) or SQLRULES (STD) is in effect or whether the table space was explicitly or implicitly created. The new LOB objects inherit the buffer pool attribute and authorization from the existing LOB objects.

CCSID *ccsid-value*

Identifies the CCSID value to be used for the table space. *ccsid-value* must identify a CCSID value that is compatible with the current value of the CCSID for the table space. See "Notes" on page 850 for a list that shows the CCSID to which a given CCSID can be changed and details about changing it.

Do not specify CCSID for a LOB table space, a table space that is implicitly created for an XML column, or a table space in a work file database.

The CCSID of a table space cannot be changed if any of the following conditions are true:

- The table space contains any table that has an index that contains expressions.
- The table space contains a system-period temporal table or a history table.
- The table space contains an archive-enabled table or an archive table.
- The table space contains an EBCDIC table with a Unicode column

CLOSE

When the limit on the number of open data sets is reached, specifies the priority in which data sets are closed.

YES
Eligible for closing before CLOSE NO data sets. This is the default unless the table space is in a work file database.

NO Eligible for closing after all eligible CLOSE YES data sets are closed.

For a table space in a work file database, DB2 uses CLOSE NO regardless of the value specified

COMPRESS
Specifies whether data compression applies to the rows of the table space or partition. Do not specify COMPRESS for a LOB table space or a table space in a work file database.

YES
Specifies data compression. The rows are not compressed until the LOAD or REORG utility is run on the table in the table space or partition, or until an insert operation is performed through the INSERT statement or the MERGE statement.

NO Specifies no data compression. Inserted rows will not be compressed. Updated rows will be decompressed. The dictionary used for compression will be erased when the LOAD REPLACE, LOAD RESUME NO, or REORG utility is run. See *DB2 Performance Monitoring and Tuning Guide* for more information about the dictionary and data compression.

DROP PENDING CHANGES
Drops pending changes to the definition of the table space and any objects within the table space. Pending changes to the definition of the table space or any object within the table space must exist.

When the DROP PENDING CHANGES clause is specified, no other options are allowed in the same ALTER TABLESPACE statement.

The DROP PENDING CHANGES clause also resets advisory REORG-pending (AREOR) status except for tables that are converting to hash access.

DSSIZE *integer* **G**
A value, in gigabytes, that indicates the maximum size for each partition, or for a LOB table space, the maximum size of each data set. DSSIZE can only be specified for a universal table space or a LOB table space. When DSSIZE is specified, no other options are allowed in the same ALTER TABLESPACE statement.

The following are valid values for *integer*:

integer **value**	**Meaning**
1	1 gigabyte
2	2 gigabytes
4	4 gigabytes
8	8 gigabytes
16	16 gigabytes
32	32 gigabytes
64	64 gigabytes
128G	128 gigabytes

256G 256 gigabytes

If *integer* is greater than 4, the data sets for the table space must be associated with a DFSMS data class that has been specified with an extended format and extended addressability.

If the table space is a partition-by-growth universal table space, the DSSIZE value must be valid depending on the values that are in effect for the MAXPARTITIONS option and the page size of the table space.

If the table space is a partition by range universal table space, the DSSIZE value must be valid depending on the values that are in effect for the current number of partitions and the page size of the table space.

The DSSIZE value must be valid depending on the maximum PIECESIZE of any associated non-partitioned secondary indexes.

The change to the DSSIZE is a pending change to the definition of the table space if the data sets of the table space are already created and if one of the following conditions is true:

- Pending definition changes already exist for the table space or any associated indexes.
- The specified DSSIZE is different than the value that is currently being used for the table space.

Otherwise, the change takes effect immediately.

If the change is a pending change to the definition of the table space, the changes are not reflected in the definition or data at the time the ALTER TABLESPACE statement is issued. Instead, the entire table space is placed in an advisory REORG-pending state (AREOR). A subsequent reorganization of the entire table space will apply the pending definition changes to the definition and data of the table space.

If the table space is a partition-by-growth universal table space with the pending DSSIZE change is applied, the number of partitions is determined based on the amount of existing data in the table space and the new DSSIZE value. Changing the DSSIZE value to be smaller might cause automatic growth of additional partitions. If LOB columns exist, additional LOB table spaces and auxiliary objects are implicitly created for the newly-grown partitions independently of whether SQLRULES(DB2) or SQLRULES(STD) is in effect or whether the table space was explicitly or implicitly created. The new LOB objects inherit the buffer pool attribute and authorization from the existing LOB objects.

LOCKMAX

Specifies the maximum number of page, row, or LOB locks an application process can hold simultaneously in the table space. If a program requests more than that number, locks are escalated. The page, row, or LOB locks are released and the intent lock on the table space or segmented table is promoted to S or X mode. If you specify LOCKMAX a for table space in a work file database, DB2 ignores the value because these types of locks are not used.

integer

Specifies the number of locks allowed before escalating, in the range 0 to 2 147 483 647.

Zero (0) indicates that the number of locks on the table or table space are not counted and escalation does not occur.

SYSTEM
Indicates that the value of field LOCKS PER TABLE(SPACE) on installation panel DSNTIPJ specifies the maximum number of page, row, or LOB locks a program can hold simultaneously in the table or table space.

If you change LOCKSIZE and omit LOCKMAX, the following results occur:

LOCKSIZE	Resultant LOCKMAX
TABLESPACE or TABLE	0
PAGE, ROW, or LOB	Unchanged
ANY	SYSTEM

If the lock size is TABLESPACE or TABLE, LOCKMAX must be omitted, or its operand must be 0.

LOCKSIZE
Specifies the size of locks used within the table space and, in some cases, also the threshold at which lock escalation occurs. Do not specify LOCKSIZE for a table space in a work file database.

ANY
Specifies that DB2 can use any lock size.

In most cases, DB2 uses LOCKSIZE PAGE LOCKMAX SYSTEM for non-LOB table spaces and LOCKSIZE LOB LOCKMAX SYSTEM for LOB table spaces. However, when the number of locks acquired for the table space exceeds the maximum number of locks allowed for a table space (an installation parameter), the page or LOB locks are released and locking is set at the next higher level. If the table space is segmented, the next higher level is the table. If the table space is not segmented, the next higher level is the table space.

TABLESPACE
Specifies table space locks.

TABLE
Specifies table locks. Use TABLE only for a segmented table space. Do not use TABLE for a universal table space.

PAGE
Specifies page locks. Do not use PAGE for a LOB table space.

ROW
Specifies row locks. Do not use ROW for a LOB table space.

LOB
Specifies LOB locks. Use LOB only for a LOB table space.

Let S denote an SQL statement that refers to a table in the table space:

- The LOCKSIZE change affects S if S is prepared and executed after the change. This includes dynamic statements and static statements that are not bound because of VALIDATE(RUN).
- If the size specified by the new LOCKSIZE is greater than the size of the old LOCKSIZE, the change affects S if S is a static statement that is executed after the change.

  The hierarchy of lock sizes, starting with the largest, is as follows:
  – table space lock
  – table lock (only for segmented table spaces)

– page lock, row lock, and LOB lock (which are at the same level)

LOGGED or NOT LOGGED

Specifies whether changes that are made to the data in the specified table space are recorded in the log.

LOGGED

Specifies that changes that are made to the data in the specified table space are recorded in the log. This applies to all tables in the specified table space and to all indexes of those tables. Table spaces and indexes that are created for XML columns inherit the logging attribute from the associated base table space. Auxiliary indexes inherit the logging attribute from the associated base table space. This can affect the logging attribute of associated LOB table spaces. See "Notes" on page 1090 for more information.

If the base table space is in informational copy-pending status (meaning updates have been made to the table space) when you change from NOT LOGGED to LOGGED, the base table space is placed in copy-pending status. All indexes of tables in the table space are unchanged from their current state; that is, if an index is currently in informational copy-pending status, it will remain in information copy-pending status.

Specifying LOGGED for a LOB table space requires that the base table space also specifies the LOGGED parameter.

LOGGED cannot be specified for XML table spaces. The logging attribute of an XML table space is inherited from its base table space.

LOGGED cannot be specified for table spaces in DSNDB06 (the DB2 catalog) or in a work file database.

NOT LOGGED

Specifies that changes that are made to data in the specified table space are not recorded in the log. This applies to all tables in the specified table space and to all indexes of those tables. Table spaces and indexes that are created for XML columns inherit the logging attribute from the associated base table space. Auxiliary indexes inherit the logging attribute from the associated base table space. This parameter can affect the logging attribute of associated LOB table spaces. See "Notes" on page 1090 for more information.

NOT LOGGED prevents undo and redo information from being recorded in the log for the base table space; however, control information for the specified base table space will continue to be recorded in the log. For a LOB table space, changes to system pages and to auxiliary indexes are logged.

NOT LOGGED is mutually exclusive with the DATA CAPTURE CHANGES parameter of CREATE TABLE and ALTER TABLE. NOT LOGGED will not be applied to the table space if any table in the table space specifies DATA CAPTURE CHANGES.

NOT LOGGED cannot be specified for XML table spaces.

NOT LOGGED cannot be specified for table spaces in the following databases:

- DSNDB06 (the DB2 catalog)
- a work file database

MAXROWS *integer*

Specifies the maximum number of rows that DB2 will consider placing on each data page. The integer can range from 1 through 255.

The change takes effect immediately for new rows added. However, the space class settings for some pages might be incorrect and could cause unproductive page visits. It is highly recommended to reorganize the table space after altering MAXROWS.

After ALTER TABLESPACE with MAXROWS is run, the table space is placed into an advisory REORG-pending status. Run the REORG TABLESPACE utility to remove the status.

Do not specify MAXROWS for a LOB table space, a table space that is implicitly created for an XML column, a table space in a work file database, or the DB2 catalog table spaces that are listed under "SQL statements allowed on the catalog" on page 2113.

MAXPARTITIONS *integer*

Specifies that the table space is partition-by-growth. *integer* specifies the maximum number of partitions to which the table space can grow or shrink. *integer* must be in the range of 1 to 4096, depending on the value that is in effect for DSSIZE and the page size of the table space, and must not be less than the number of physical partitions that are already allocated for the table space. See "CREATE TABLESPACE" on page 1455 for more information about how DSSIZE and the page size are related.

MAXPARTITIONS can only be specified for a simple table space that contains only one table, a segmented table space that contains only one table, or a partitioned-by-growth universal table space. The table space must have DB2-managed data sets.

Although physical data sets are not defined when the MAXPARTITIONS value is issued, there can be storage and cpu overhead. If an increase in the number of partitions is expected by using the MAXPARTITONS clause, be aware that specifying an value larger than necessary, such as 4096 (the maximum value), as a default for all of your partition-by-growth table spaces can cause larger than expected storage requests.

The change to the value of MAXPARTITIONS is a pending change to the definition of the table space if the data sets of the table space are already created and one of the following conditions is true:

- Pending changes to the definition of the table space or associated indexes already exist.
- The table space is converted from a simple table space to a partition-by-growth universal table space.
- The table space is converted from a segmented table space to a partition-by-growth universal table space.
- The table space is changed to use a different page size by specifying the BUFFERPOOL option of ALTER TABLESPACE.
- The table space is changed to use a different DSSIZE by specifying the DSSIZE option of ALTER TABLESPACE.

Otherwise, the change is an immediate change.

If the change is a pending change to the definition of the table space, the changes are not reflected in the definition or data at the time the ALTER TABLESPACE statement is issued. Instead, the entire table space is placed in

an advisory REORG-pending state (AREOR). A subsequent reorganization of the entire table space will apply the pending definition changes to the definition and data of the table space.

If MAXPARTITIONS is specified on a simple or segmented table space, the table space is converted to a partition-by-growth universal table space that can grow to a maximum number of *integer* partitions. The SEGSIZE is set to the default of 32 if the SEGSIZE prior to conversion is less than 32. Otherwise, the value of SEGSIZE is inherited from the original table space. The DSSIZE is set to the default 4 gigabytes.

If the data sets of the table space are not defined, the number of partitions is set to 1 during the conversion to a partition-by-growth universal table space from a simple or segmented table space.

If the data sets of the table space are created, the number of partitions is determined based on amount of existing data at the time the pending change to the definition of the table space is applied. Partition growth can happen. If LOB columns exist, additional LOB table spaces and auxiliary objects are implicitly created for the newly-grown partitions, regardless of whether SQLRULES(DB2) or SQLRULES(STD) is in effect or whether the table space was explicitly or implicitly created. The new LOB objects inherit the buffer pool attribute and thereby authorization from the existing LOB objects.

If the table space is defined with LOCKSIZE TABLE, the lock size will be reset to LOCKSIZE TABLESPACE during conversion to a partition-by-growth universal table space.

MEMBER CLUSTER YES or MEMBER CLUSTER NO
Specifies whether the table space uses the MEMBER CLUSTER page set structure. The MEMBER CLUSTER clause can only be specified for a partition-by-growth or range-partitioned universal table space. Changing the MEMBER CLUSTER structure for a table space results in a pending definition change for the table space. The table space is placed in advisory REORG-pending state (AREOR). Running a utility like REORG with SHRLEVEL(CHANGE) or SHRLEVEL(REFERENCE) on the entire table space resets this state.

MEMBER CLUSTER YES
Specifies that the MEMBER CLUSTER page set structure is to be used for the specified table space when the table space is already defined as a partition-by-growth or range-partitioned universal table space.

MEMBER CLUSTER YES cannot be specified for LOB, workfile, or XML table spaces, or for table spaces that are organized for hash access.

MEMBER CLUSTER NO
Specifies that the table space does not use the MEMBER CLUSTER page set structure when the table space is already defined as a partition-by-growth or range-partitioned universal table space. If the universal table space is already defined to use the MEMBER CLUSTER page set structure, specifying **MEMBER CLUSTER NO** on the ALTER TABLESPACE statement removes the MEMBER CLUSTER page set structure from the table space.

MEMBER CLUSTER NO is the default.

SEGSIZE *integer*
Specifies that the table space is a universal table space, where *integer* specifies the number of pages that are to be assigned to each segment of the table space.

integer must be a multiple of 4 between 4 and 64 (inclusive). When SEGSIZE is specified, no other options are allowed in the same ALTER TABLESPACE statement.

SEGSIZE can only be specified for a universal table space or a partitioned table space that uses table-controlled partitioning.

The change to the value of SEGSIZE is a pending change to the definition of the table space if the data sets of the table space are already created and one of the following conditions is true:

- Pending changes to the definition of the table space or its associated indexes already exist.
- The specified SEGSIZE value for a universal table space is different than the existing value.
- The table space is converted from a partitioned table space to a range-partitioned universal table space.

Otherwise, the change is an immediate change.

If the change is a pending change to the definition of the table space, the changes are not reflected in the definition or data at the time the ALTER TABLESPACE statement is issued. Instead, the entire table space is placed in an advisory REORG-pending state (AREOR). A subsequent reorganization of the entire table space will apply the pending definition changes to the definition and data of the table space.

If the existing FREEPAGE value (the number of pages to be left free) is greater than or equal to the new SEGSIZE value, the number of pages is adjusted to be one less than the new SEGSIZE value.

If the table space is a partitioned table space, the partitioned table space is converted to a range-partitioned universal table space with a segment size specified by *integer*. The MEMBER CLUSTER attribute is inherited from the original table space. The number of partitions is inherited from the original table space. If the original DSSIZE attribute has a value of 0, the DSSIZE is set to the original maximum partition size. Otherwise, the DSSIZE attribute is inherited from the original table space.

If the table space is a partition-by-growth universal table space when the pending SEGSIZE change is applied, the number of partitions is determined based on the amount of existing data in the table space and the new SEGSIZE value. Changing the SEGSIZE value to be smaller might cause automatic growth of additional partitions. If LOB columns exist, additional LOB table spaces and auxiliary objects are implicitly created for the newly-grown partitions independently of whether SQLRULES(DB2) or SQLRULES(STD) is in effect or whether the table space was explicitly or implicitly created. The new LOB objects inherit the buffer pool attribute and authorization from the existing LOB objects.

TRACKMOD

Specifies whether DB2 tracks modified pages in the space map pages of the table space or partition. Do not specify TRACKMOD for a LOB table space or a table space in a work file database.

For the changed TRACKMOD option to take effect, the table space or partition needs to be stopped and restarted. The table space or partition can be stopped and restarted by running the STOP DATABASE command followed by the START DATABASE command, or by running the REORG utility on the table

space or partition. See -STOP DATABASE (DB2) (DB2 Commands) and -START DATABASE (DB2) (DB2 Commands) or REORG TABLESPACE (DB2 Utilities) for information.

YES
DB2 tracks changed pages in the space map pages to improve the performance of incremental image copy. For data sharing, changing TRACKMOD to YES causes additional SCA (shared communication area) storage to be used until after the next full or incremental image copy is taken or until TRACKMOD is set back to NO.

NO DB2 does not track changed pages in the space map pages. It uses the LRSN value in each page to determine whether a page has been changed.

FREEPAGE *integer*
Specifies how often to leave a page of free space when the table space is loaded or reorganized. One free page is left after every *integer* pages; *integer* can range from 0 to 255. FREEPAGE 0 leaves no free pages. Do not specify FREEPAGE for a LOB table space, a table space that is implicitly created for an XML column, or a table space in a work file database.

If the table space is segmented, the number of pages left free must be less than the SEGSIZE value. If the number of pages to be left free is greater than or equal to the SEGSIZE value, then the number of pages is adjusted downward to one less than the SEGSIZE value.

This change to the description of the table space or partition has no effect until data in the table space or partition is loaded or reorganized. For XML table spaces, this change has no effect until data in the table space is reorganized.

Related information:

Reserving free space for table spaces (DB2 Performance)

Reserving free spaces for indexes (DB2 Performance)

PCTFREE *smallint*
Specifies what percentage of each page to leave as free space when the table space is loaded or reorganized. The default value is PCTFREE 5, which specifies that 5% of the space on each data page is reserved as free space. The first record on each page is loaded without restriction. When additional records are loaded, at least *integer* percent of free space is left on each page. *integer* can range from 0 to 99. Do not specify PCTFREE for a LOB table space, a table space that is implicitly created for an XML column, or a table space in a work file database.

FOR UPDATE *smallint*
Specifies the percentage of space to reserve as free space on each page, for use by subsequent UPDATE operations, when data is added to the table by INSERT operations or utilities. The *smallint* value is an integer in the range -1 to 99. FOR UPDATE -1 specifies that 5% of free space is reserved initially, and the amount of free spaces is calculated automatically based on certain real-time statistics values. The first record on each page is loaded always loaded without restriction.

If this value is not specified, the value of the PCTFREE_UPD subsystem parameter is used.

The value is recorded in the PCTFREE_UPD column of the SYSIBM.SYSTABLEPART catalog table.

The FOR UPDATE *smallint* values do not apply to LOB table spaces, XML table spaces, or table spaces that use hash organization.

The sum of the values for PCTFREE *smallint* and FOR UPDATE*smallint* must be less than or equal to 99.

If FOR UPDATE*smallint* is not specified and the sum of PCTFREE *smallint* and the PCTFREE_UPD subsystem parameter value is greater than or equal to 99, DB2 uses a smaller value for PCTFREE_UPD.

This change to the description of the table space or partition has no effect until data in the table space or partition is loaded or reorganized. For XML table spaces, this change has no effect until data in the table space is reorganized.

Related information:

Reserving free space for table spaces (DB2 Performance)

Reserving free spaces for indexes (DB2 Performance)

USING

Specifies whether a data set for the table space or partition is managed by the user or is managed by the DB2 system. If the table space is partitioned, USING applies to the data set for the partition that is identified in the PARTITION clause. If the table space is a partition-by-growth table space, USING can only be specified at the table space level. If the table space is not partitioned, USING applies to every data set that is eligible for the table space. (A nonpartitioned table space can have more than one data set if PRIQTY+118 × SECQTY is at least 2 gigabytes.)

If the USING clause is specified, the table space or partition must be in the stopped state when the ALTER TABLESPACE statement is executed. See Altering storage attributes to determine how and when changes take effect. Do not specify the USING clause if the table space is in a work file database.

VCAT *catalog-name*

Specifies a user-managed data set with a name that starts with *catalog-name*. The VCAT clause must not be specified if the table space is a partition-by-growth table space. You must specify the catalog name in the form of an SQL identifier. You must specify an alias[25] if the name of the integrated catalog facility catalog is longer than eight characters. When the new description of the table space is applied, the integrated catalog facility catalog must contain an entry for the data set that conforms to the DB2 naming conventions set forth in *DB2 Administration Guide*.

One or more DB2 subsystems could share integrated catalog facility catalogs with the current server. To avoid the chance of having one of those subsystems attempt to assign the same name to different data sets, select a value for *catalog-name* that is not used by the other DB2 subsystems.

STOGROUP *stogroup-name*

Specifies a DB2-managed data set that resides on a volume of the identified storage group. *stogroup-name* must identify a storage group that exists at the current server and the privilege set must include SYSADM authority, SYSCTRL authority, or the USE privilege for the storage group. When the new description of the table space is applied, the description of the storage group must include at least one volume serial number, each

25. The alias of an integrated catalog facility catalog

volume serial number must identify a volume that is accessible to z/OS for dynamic allocation of the data set, and all identified volumes must be of the same device type. Furthermore, the integrated catalog facility catalog used for the storage group must not contain an entry for the data set.

If you specify USING STOGROUP and the current data set for the table space or partition is managed by DB2:

- Omission of the PRIQTY clause is an implicit specification of the current PRIQTY value.
- Omission of the SECQTY clause is an implicit specification of the current SECQTY value.
- Omission of the ERASE clause is an implicit specification of the current ERASE rule.

If you specify USING STOGROUP to convert from user-managed data sets to DB2-managed data sets:

- Omission of the PRIQTY clause is an implicit specification of the default value. For information on how DB2 determines the default value, see Rules for primary and secondary space allocation.
- Omission of the SECQTY clause is an implicit specification of the default value. For information on how DB2 determines the default value, see Rules for primary and secondary space allocation.
- Omission of the ERASE clause is an implicit specification of ERASE NO.

PRIQTY *integer*

Specifies the minimum primary space allocation for a DB2-managed data set of the table space or partition. This clause can be specified only if the data set is managed by DB2, and if one of the following is true:

- USING STOGROUP is specified.
- A USING clause is not specified.

If PRIQTY is specified (with a value other than -1), the primary space allocation is at least *n* kilobytes, where *n* is the value of *integer* with the following exceptions:

- For 4 KB page sizes, if *integer* is less than 12, *n* is 12.
- For 8 KB page sizes, if *integer* is less than 24, *n* is 24.
- For 16 KB page sizes, if *integer* is less than 48, *n* is 48.
- For 32 KB page sizes, if *integer* is less than 96, *n* is 96.
- For any page size, if *integer* is greater than 67108864, *n* is 67108864.

For LOB table spaces, the exceptions are:

- For 4 KB pages sizes, if *integer* is less than 200, *n* is 200.
- For 8 KB pages sizes, if *integer* is less than 400, *n* is 400.
- For 16 KB pages sizes, if *integer* is less than 800, *n* is 800.
- For 32 KB pages sizes, if *integer* is less than 1600, *n* is 1600.
- For any page size, if *integer* is greater than 4194304, *n* is 4194304.

The maximum value allowed for PRIQTY is 64GB (67108864 kilobytes).

If PRIQTY -1 is specified, DB2 uses a default value for the primary space allocation. For information on how DB2 determines the default value for primary space allocation, see Rules for primary and secondary space allocation.

If PRIQTY is omitted and USING STOGROUP is specified, the value of PRIQTY is its current value. (However, if the current data set is being changed from being user-managed to DB2-managed, the value is its default value. See the description of USING STOGROUP.)

If you specify PRIQTY and do not specify a value of -1, DB2 specifies the primary space allocation to access method services using the smallest multiple of *p* KB not less than *n*, where *p* is the page size of the table space. The allocated space can be greater than the amount of space requested by DB2. For example, it could be the smallest number of tracks that will accommodate the request. To more closely estimate the actual amount of storage, see DEFINE CLUSTER command (DFSMS Access Method Services for Catalogs).

At least one of the volumes of the identified storage group must have enough available space for the primary quantity. Otherwise, the primary space allocation will fail.

See Altering storage attributes to determine how and when changes to PRIQTY take effect.

SECQTY *integer*
Specifies the minimum secondary space allocation for a DB2-managed data set of the table space or partition. This clause can be specified only if the data set is managed by DB2, and if one of the following is true:
- USING STOGROUP is specified.
- A USING clause is not specified.

If SECQTY -1 is specified, DB2 uses a default value for the secondary space allocation.

If USING STOGROUP is specified and SECQTY is omitted, the value of SECQTY is its current value. (However, if the current data set is being changed from being user-managed to DB2-managed, the value is its default value. See the description of USING STOGROUP.)

For information on the actual value that is used for secondary space allocation, whether you specify a value or DB2 uses a default value, see Rules for primary and secondary space allocation.

If you specify SECQTY and do not specify a value of -1, DB2 specifies the secondary space allocation to access method services using the smallest multiple of *p* KB not less than *n*, where *p* is the page size of the table space. The allocated space can be greater than the amount of space requested by DB2. For example, it could be the smallest number of tracks that will accommodate the request. To more closely estimate the actual amount of storage, see the description of the DEFINE CLUSTER command (DFSMS Access Method Services for Catalogs) for z/OS DFSMS Access Method Services for catalogs.

See Altering storage attributes to determine how and when changes to SECQTY take effect.

ERASE
Indicates whether the DB2-managed data sets for the table space or partition are to be erased before they are deleted during the execution of a utility or an SQL statement that drops the table space.

NO Does not erase the data sets. Operations involving data set deletion will perform better than ERASE YES. However, the data is still accessible, though not through DB2.

YES
Erases the data sets. As a security measure, DB2 overwrites all data in the data sets with zeros before they are deleted.

This clause can be specified only if the data set is managed by DB2, and if one of the following is true:

- USING STOGROUP is specified.
- A USING clause is not specified.

If you specify ERASE, the table space or partition must be in the stopped state when the ALTER TABLESPACE statement is executed. If you specify ERASE for a partitioned table space, you must also specify the ALTER PARTITION clause. See Altering storage attributes to determine how and when changes take effect.

GBPCACHE

In a data sharing environment, specifies what pages of the table space or partition are written to the group buffer pool in a data sharing environment. In a non-data-sharing environment, you can specify GBPCACHE for a table space other than one in a work file database, but it is ignored. Do not specify GBPCACHE for a table space in a work file database in either environment (data sharing or not). In addition, you cannot alter the GBPCACHE value of some DB2 catalog table spaces; for a list of these table spaces, see "SQL statements allowed on the catalog" on page 2113.

CHANGED

When there is inter-DB2 R/W interest on the table space or partition, updated pages are written to the group buffer pool. When there is no inter-DB2 R/W interest, the group buffer pool is not used. Inter-DB2 R/W interest exists when more than one member in the data sharing group has the table space or partition open, and at least one member has it open for update.

If the table space is in a group buffer pool that is defined to be used only for cross-invalidation (GBPCACHE NO), CHANGED is ignored and no pages are cached to the group buffer pool.

ALL

Indicates that pages are to be cached in the group buffer pool as they are read in from DASD.

Exception: In the case of a single updating DB2 when no other DB2 subsystems have any interest in the page set, no pages are cached in the group buffer pool.

If the table space is in a group buffer pool that is defined to be used only for cross-invalidation (GBPCACHE NO), ALL is ignored and no pages are cached to the group buffer pool.

SYSTEM

Indicates that only changed system pages within the LOB table space are to be cached to the group buffer pool. A system page is a space map page or any other page that does not contain actual data values.

Use SYSTEM only for a LOB table space.

NONE

Indicates that no pages are to be cached to the group buffer pool. DB2 uses the group buffer pool only for cross-invalidation.

If you specify NONE, the table space or partition must not be in recover pending status when the ALTER TABLESPACE statement is executed.

If you specify GBPCACHE in a data sharing environment, the table space or partition must be in the stopped state when the ALTER TABLESPACE statement is executed.

ALTER PARTITION *integer*

Specifies that the identified partition of the table space is to be changed. For a table space that has *n* partitions, you must specify an integer in the range 1 to *n*. You must not use this clause for a nonpartitioned table space, for a LOB table space, or a partition-by-growth table space. At least one of the following clauses must be specified:

- COMPRESS
- ERASE
- FREEPAGE
- GBPCACHE
- PCTFREE
- PRIQTY
- SECQTY
- TRACKMOD
- USING

The changes specified by these clauses affect only the identified partition.

Do not specify the following clauses for ALTER PARTITION for partitions of a table space that is implicitly created for an XML column.

- CCSID
- FREEPAGE
- MAXROWS
- PCTFREE

Notes

Running utilities:

You cannot execute the ALTER TABLESPACE statement while a DB2 utility has control of the table space.

Altering more than one partition:

To change FREEPAGE, PCTFREE, USING, PRIQTY, SECQTY, COMPRESS, ERASE, or GBPCACHE for more than one partition, you must use separate ALTER TABLESPACE statements.

Altering storage attributes:

The USING, PRIQTY, SECQTY, and ERASE clauses define the storage attributes of the table space or partition. If you specify USING or ERASE when altering storage attributes, the table space or partition must be in the stopped state when the ALTER TABLESPACE statement is executed. You can use a STOP DATABASE...SPACENAM... command to stop the table space or partition.

If the catalog name changes, the changes take effect after you move the data and start the table space or partition using the START DATABASE...SPACENAM... command. The catalog name can be implicitly or explicitly changed by the ALTER TABLESPACE statement. The catalog name also changes when you move the data to a different device. See the procedures for moving data in *DB2 Administration Guide*.

Changes to the secondary space allocation (SECQTY) take effect the next time DB2 extends the data set; however, the new value is not reflected in the integrated catalog until you use the REORG, RECOVER, or LOAD REPLACE utility on the table space or partition. The changes to the other storage attributes take effect the next time the page set is reset. For a non-LOB table space, the page set is reset when you use the REORG, RECOVER, or LOAD REPLACE utilities on the table space or partition. For a LOB table space, the page set is reset when RECOVER is run on the LOB

table space or LOAD REPLACE is run on its associated base table space. If there is not enough storage to satisfy the primary space allocation, a REORG might fail. If you change the primary space allocation parameters or erase rule, you can have the changes take effect earlier if you move the data before you start the table space or partition.

Recommended GBPCACHE setting for LOB table spaces:
For LOB table spaces, use the GBPCACHE CHANGED option instead of the GBPCACHE SYSTEM option. Due to the usage patterns of LOBs, the use of GBPCACHE CHANGED can help avoid excessive and synchronous writes to disk and the group buffer pool.

Altering table spaces for tables that use hash organization:
Certain attributes of the table space, such as buffer pool and page size, might affect performance of tables that use hash organization. Changes related to the hash organization of a table will be validated and might generate error messages as described in "CREATE TABLE" on page 1388 and "ALTER TABLE" on page 984.

Altering the logging attribute of a table space:
If the logging attribute (specified with the LOGGED or NOT LOGGED parameter) of a table space is altered frequently, the size of SYSIBM.SYSCOPY might need to be increased.

The logging attribute of the table space cannot be altered if the table space has been updated in the same unit of recovery.

A full image copy of the table space should be taken:

- Before altering a table space to NOT LOGGED
- After altering a table space to LOGGED

If a table space has data changes after an image copy is taken (the table space is in informational COPY-pending state), and the table space is altered from NOT LOGGED to LOGGED, the table space is marked COPY-pending and a full image copy of the table space must be taken.

An XML table space with the LOGGED logging attribute has its logging attribute altered to NOT LOGGED when the logging attribute of the associated base table space is altered from LOGGED to NOT LOGGED. When this happens, the logging attribute of the XML table space is said to be *linked* to the logging attribute of the base table space. When the logging attribute of the base table space is altered back to LOGGED, all logging attributes that are linked for the associated XML table spaces are altered back to LOGGED, and all of these links are dissolved.

A LOB table space with the LOGGED logging attribute has its logging attribute altered to NOT LOGGED when the logging attribute of the associated base table space is altered from LOGGED to NOT LOGGED. When this happens, the logging attribute of the LOB table space is said to be *linked* to the logging attribute of the base table space. When the logging attribute of the base table space is altered back to LOGGED, all logging attributes that are linked for the associated LOB table spaces are altered back to LOGGED, and all of these links are dissolved.

You can dissolve the link between these logging attributes by altering the logging attribute of the LOB table space to NOT LOGGED, even though it has already been implicitly given this logging attribute. After such an alter, the logging attribute of the LOB table space is unaffected when the logging attribute of the base table is altered back to LOGGED. A LOB table space with the NOT LOGGED logging attribute does not have this attribute

changed in any way if the logging attribute of the associated base table space is altered from LOGGED to NOT LOGGED. When altered in this way, the logging attributes of the LOB table space and the base table space are not linked. If the base table space is altered back to LOGGED, the logging attribute of any LOB table spaces that are not linked to the logging attribute of the base table space remain unchanged.

Altering table spaces for DB2 catalog tables:
For details on altering options on catalog tables, see "SQL statements allowed on the catalog" on page 2113.

Invalidation of packages:
All of the packages that refer to that table space are invalidated when any of the following conditions are true:

- The SBCS CCSID attribute of a table space is changed.
- When changing the MAXPARTITIONS attribute of a table space.
- The SEGSIZE attribute of a partitioned table space is changed to convert the table space to a range-partitioned universal table space.

Pending changes to the definition of a table space:
Issuing the ALTER TABLESPACE statement with certain options can cause a pending change to the definition of a table space. When an ALTER TABLESPACE statement that causes pending changes to the definition is executed, semantic validation and authorization checking are performed. However, changes to the table space definition and data are not applied and the table space is placed in advisory REORG-pending state (AREOR). The pending changes are recorded in the SYSIBM.SYSPENDINGDDL catalog table. The REORG utility that specifies SHRLEVEL CHANGE or REFERENCE should be run on the table space to apply the pending changes to the definition and data of the table space. When the pending changes are applied, dependent packages are invalidated, the corresponding entries in the SYSIBM.SYSPENDINGDDL catalog table are removed, and the advisory REORG-pending state is removed.

The following ALTER TABLESPACE options can cause pending changes to the definition of the table space under certain conditions:

- BUFFERPOOL
- DSSIZE
- MAXPARTITIONS
- SEGSIZE

The changes that are caused by all other options occur when the ALTER TABLESPACE statement is executed.

Restrictions on ALTER TABLESPACE statements that cause pending changes:
ALTER TABLESPACE statements that cause pending changes have the following restrictions:

- Options that cause pending changes cannot be specified with options that take effect immediately
- Options that cause pending changes cannot be specified for the following objects:
  - The catalog
  - System objects
  - Objects in a workfile database
- The DROP PENDING CHANGES clause cannot be specified for a catalog table space

- If there are pending changes to the table space, you cannot use ALTER TABLESPACE to change from a DB2-managed data set to a user-managed data set
- If there are pending changes to the table space, you cannot specify the following clauses:
  - FREEPAGE
  - ALTER PARTITION FREEPAGE
- If there are pending changes to the table space, or to tables contained in the table space, you cannot specify the CCSID clause
- If the table space, or any table it contains is in an incomplete state, you cannot specify options that cause pending changes

Alternative syntax and synonyms:

For compatibility with previous releases of DB2, the following keywords are supported:

- You can specify the LOCKPART clause, but it has no effect. DB2 treats all partitioned table spaces as if they were defined as LOCKPART YES. LOCKPART YES specifies the use of selective partition locking. When all the conditions for selective partition locking are met, DB2 locks only the partitions that are accessed. When the conditions for selective partition locking are not met, DB2 locks every partition of the table space.
- When altering the partitions of a partitioned table space, the ALTER keyword that precedes PARTITION keyword is optional and if ALTER keyword is omitted, then you can specify PART as a synonym for PARTITION.
- You can specify LOG YES as a synonym for LOGGED and LOG NO as a synonym for NOT LOGGED.

Examples

Example 1: Alter table space DSN8S11D in database DSN8D11A. BP2 is the buffer pool associated with the table space. PAGE is the level at which locking is to take place.

```
ALTER TABLESPACE DSN8D11A.DSN8S11D
  BUFFERPOOL BP2
  LOCKSIZE PAGE;
```

Example 2: Alter table space DSN8S11E in database DSN8D11A. The table space is partitioned. Indicate that the data sets of the table space are not to be closed when there are no current users of the table space. Also, change all of the partitions so that DB2 will use a formula to determine any secondary space allocations, and change partition 1 to use a PCTFREE value of 20.

```
ALTER TABLESPACE DSN8D11A.DSN8S11E
  CLOSE NO
  SECQTY -1
  ALTER PARTITION 1 PCTFREE 20;
```

Example 3: The following statement changes the maximum number of partitions in a partition-by-growth table space:

```
ALTER TABLESPACE TS01DB.TS01TS
  MAXPARTITIONS 30;
```

ALTER TRIGGER

The ALTER TRIGGER statement changes the description of a trigger at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set that is defined below must include at least one of the following:

- Ownership of the trigger
- The ALTERIN privilege on the schema
- SYSADM authority
- SYSCTRL authority
- System DBADM

The authorization ID that matches the schema name implicitly has the ALTERIN privilege on the schema.

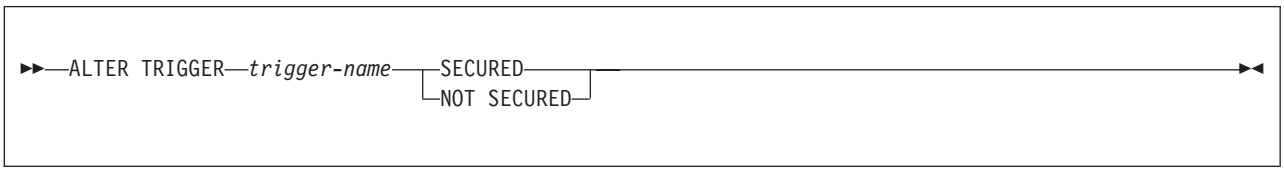
Privilege set: If the statement is embedded in an application program, the privilege set is the set of privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set is the set of privileges that are held by the SQL authorization ID of the process. The specified trigger name can include a schema name (a qualifier). If the schema name is not the same as one of the authorization ID of the process, one of the following conditions must be met:

- The privilege set includes SYSADM or SYSCTRL authority.
- The authorization ID of the process has the ALTERIN privilege on the schema.

At least one of the following privileges is required if the SECURED option is specified:

- SECADM authority
- CREATE_SECURE_OBJECT privilege

Syntax



Description

trigger-name
Identifies the trigger that is to be changed. The trigger must exist at the current server.

SECURED or NOT SECURED
Specifies that the trigger is to be changed to be secure or not secure. Changing

a trigger between SECURED and NOT SECURED causes an implicit rebind of the trigger package. If an error is encountered during the implicit rebind of the trigger package, the ALTER TRIGGER statement returns the error.

SECURED
Specifies the trigger is considered secure.

SECURED must be specified for a trigger if its subject table is using row access control or column access control. SECURED must also be specified for a trigger that is created for a view and one or more of the underlying tables in the view definition is using row access control or column access control.

NOT SECURED
Specifies the trigger is considered not secure.

NOT SECURED must not be specified for a trigger whose subject table is using row access control or column access control. NOT SECURED must also not be specified for a trigger that is created for a view and one or more of the underlying tables in the view definition is using row access control or column access control.

Notes

Changing a trigger from NOT SECURED to SECURED:
Typically, the security administrator will examine the data that is accessed by a trigger, ensure that it is secure, and grant the CREATE_SECURE_OBJECT privilege to the owner of the trigger. After the trigger is changed to SECURED, the security administrator will revoke the CREATE_SECURE_OBJECT privilege from the owner of the trigger.

The trigger is considered secure after the ALTER TRIGGER statement is executed. DB2 treats the SECURED attribute as an assertion that declares that the user has established an audit procedure for all activities in the trigger body. If a secure trigger references user-defined functions, DB2 assumes those functions are secure without validation. If those functions can access sensitive data, the user with SECADM authority needs to ensure that those functions are allowed to access that data and that an audit procedure is in place for all versions of those functions, and that all subsequent ALTER FUNCTION statements or changes to external packages are being reviewed by this audit process.

A trigger must be secure if its subject table is using row access control or column access control. SECURED must also be specified for a trigger that is created for a view and one or more of the underlying tables in the view definition is using row access control or column access control.

Altering a trigger from SECURED to NOT SECURED:
The ALTER TRIGGER statement returns an error if the subject table of the trigger is using row access control or column access control, or if the trigger is for a view and one or more of the underlying tables in the view definition is using row access control or column access control.

ALTER TRIGGER statement and implicit rebind:
The trigger package is implicitly rebound when the trigger is changed by using the ALTER TRIGGER statement. No additional BIND related privileges are required for this implicit rebind. If an error is encountered during the implicit rebind, the ALTER TRIGGER statement fails and returns the error.

Row access control and column access control that is not enforced for transition variables and transition tables:

If row access control or column access control is enforced for the subject table of the trigger, row permissions and column masks are not applied to the initial values of transition variables and transition tables. Row access control and column access control is enforced for the triggering table, but is ignored for transition variables and transition tables that are referenced in the body of the trigger body or are passed as arguments to user-defined functions that are invoked in the body of the trigger. To ensure that there are no security concerns for SQL statements accessing sensitive data in transition variables and transition tables in the trigger action, the trigger must be changed to use the SECURED option. If a trigger is not secure, row access control and column access control cannot be enforced for the triggering table.

Examples

Example 1: Change the definition of trigger TRIGGER1 to secured:

```
ALTER TRIGGER TRIGGER1
 SECURED;
```

ALTER TRUSTED CONTEXT

The ALTER TRUSTED CONTEXT statement modifies the definition of a trusted context at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

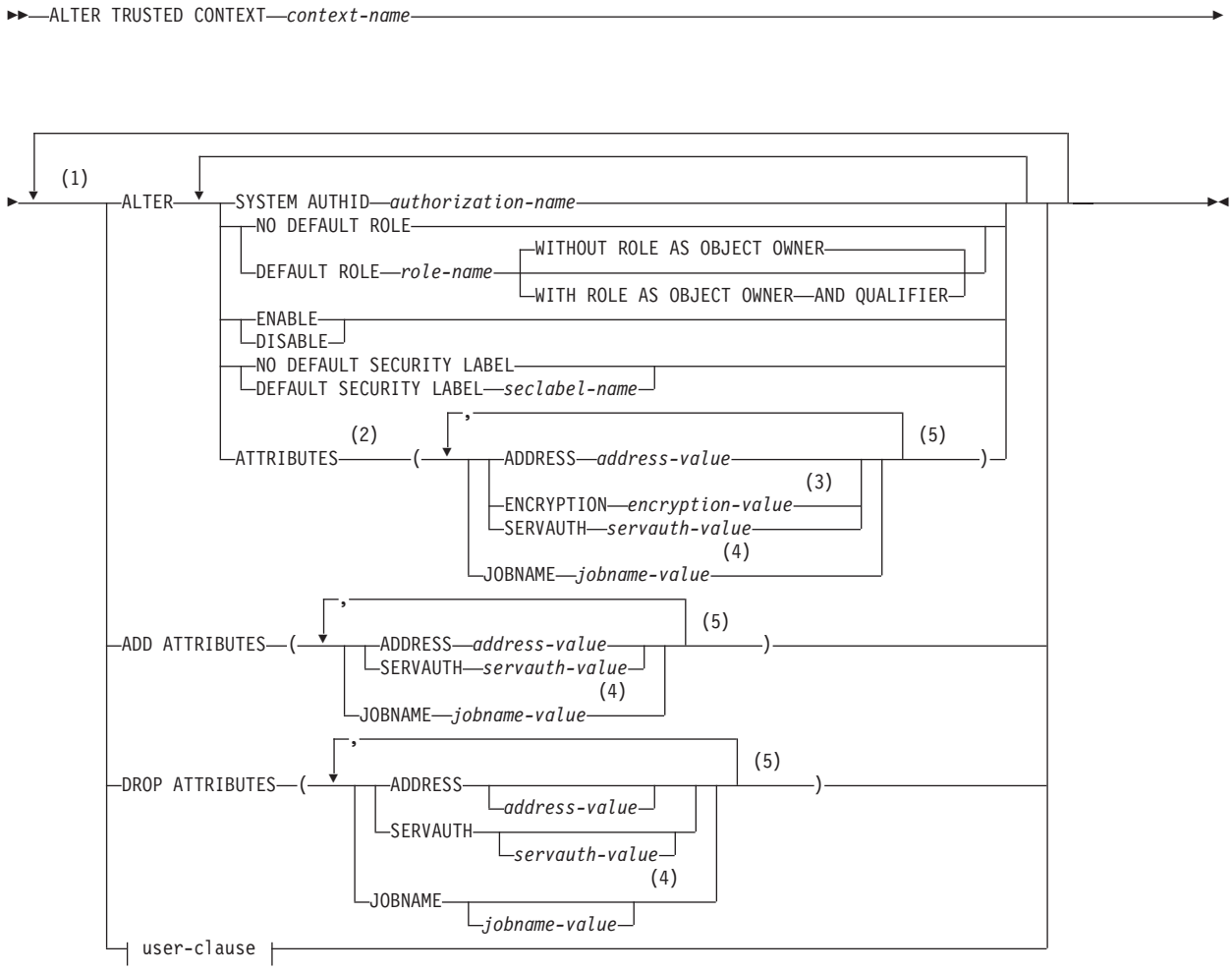
The privilege set that is defined below must include at least one of the following:

- SYSADM authority
- SECADM authority

Privilege set: If the statement is embedded in an application program, the privilege set is the set of privileges that are held by the owner of the plan or package.

If the statement is dynamically prepared, the privilege set is the union of the set of privileges that are held by each authorization ID of the process. If the statement is run in a trusted context with a role, the privilege set is the union of the set of privileges that are held by the role that is associated with the primary authorization ID and the set of privileges that are held by each authorization ID of the process.

Syntax



Notes:

1 These clauses can be specified in any order. Each clause must not be specified more than one time.

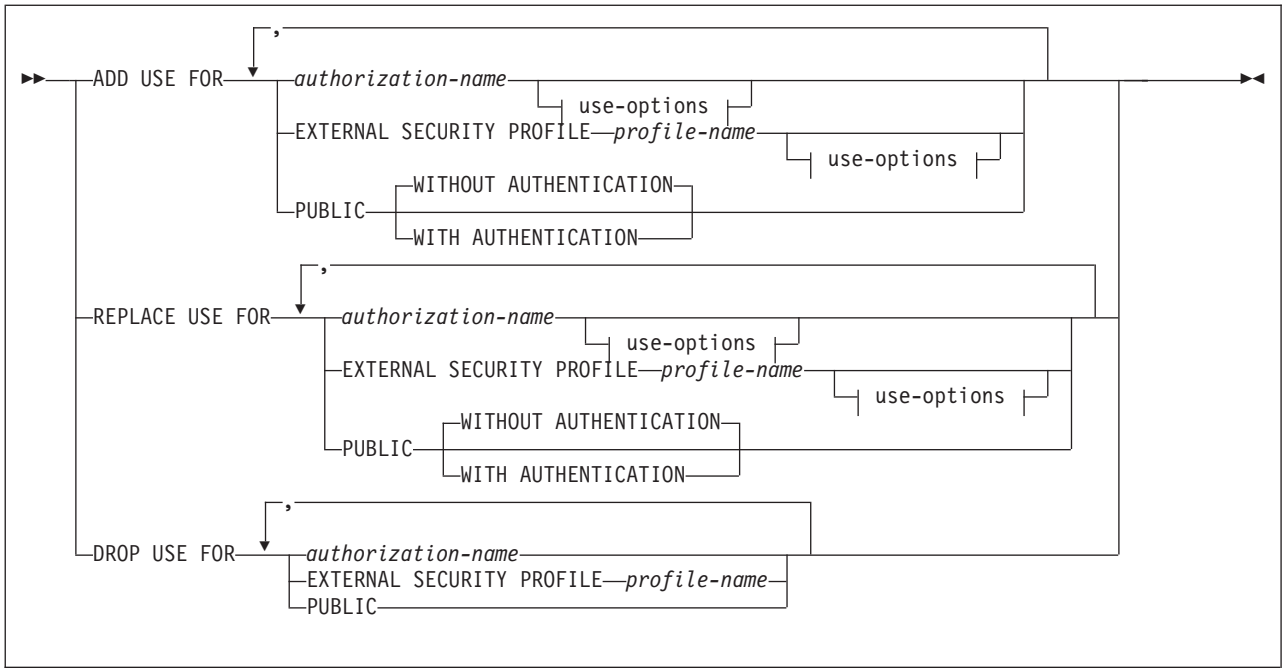
2 This clause and the clauses that follow can be specified in any order. Each clause must not be specified more than one time.

3 ENCRYPTION must not be specified more than one time.

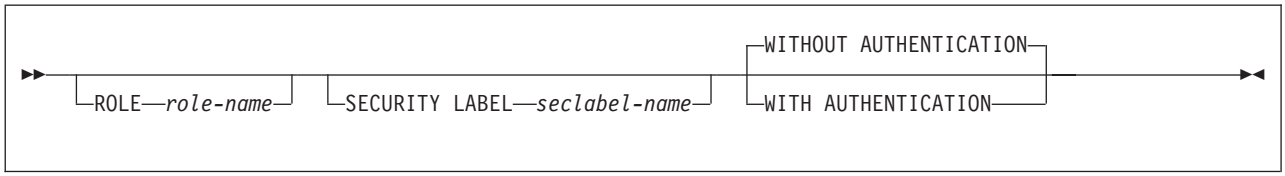
4 JOBNAME must not be specified with ADDRESS, ENCRYPTION, or SERVAUTH.

5 Each pair of attribute name and corresponding value must be unique.

user-clause:



use-options:



Description

context-name
Identifies the trusted context to alter. *context-name* must refer to a trusted context that exists at the current server.

ALTER
Specifies that changes are to be made to the definition of an existing trusted context.

SYSTEM AUTHID *authorization-name*
Specifies that *authorization-name* is the system authorization ID for the trusted context. The system authorization ID is the primary authorization ID of the DB2 system that establishes the connection. For a remote connection, the authorization ID is derived from the system used ID that is provided by the external entity, such as a middleware server. For a local connection, the system authorization ID is derived depending on the sources, as specified in Table 107.

Table 107. System authorization ID for a local connection

Source of local connection	System authorization ID
Started task (RRSAF)	USER parameter on JOB statement or RACF USER.
TSO	TSO logon ID
BATCH	USER parameter on JOB statement

authorization-name must not be associated with an existing trusted context.

NO DEFAULT ROLE or DEFAULT ROLE *role-name*
Specifies whether a default role is associated with a trusted connection that is based on the specified trusted context. If a trusted connection for the specified context is active, the change goes into effect at the next connection reuse attempt or when a new connection is requested.

NO DEFAULT ROLE
Specifies that the trusted context does not have a default role. The authorization ID of the process is the owner of any object that is created using a trusted connection that is based on this trusted context. That authorization ID must possess all of the privileges that are necessary to create that object.

DEFAULT ROLE *role-name*
Specifies that *role-name* is the role for the trusted context. *role-name* must identify a role that exists at the current server. This role is used with the user in a trusted connection that is based on the specified trusted context when the user does not have a user-specified role that is defined as part of the definition of this trusted context.

WITHOUT ROLE AS OBJECT OWNER or WITH ROLE AS OBJECT OWNER AND QUALIFIER
Specifies whether a role is used as the owner of objects that are created using a trusted connection that is based on the specified trusted context. If a trusted connection for the specified context is active, the change goes into effect at the next connection reuse attempt or when a new connection is requested.

WITHOUT ROLE AS OBJECT OWNER
Specifies that a role is not used as the owner of the objects that are created using a trusted connection that is based on the specified trusted context. The authorization ID of the process is the owner of any object that is the created using a trusted connection that is based on this trusted context. That authorization ID must possess all of the privileges that are necessary to create the object.

WITHOUT ROLE AS OBJECT OWNER is the default.

WITH ROLE AS OBJECT OWNER AND QUALIFIER
Specifies that the context assigned role is the owner of the objects that are created using a trusted connection that is based on this trusted context. That role must possess all of the privileges that are necessary to create the object. The context assigned role is the role that is defined for the user within this trusted context, if one is defined. Otherwise, the role is the default role that is associated with the trusted context. The role is also used as the grantor for any GRANT statements that are issued, and the revoker for any REVOKE statement that are issued using a trusted connection that is based on this trusted context.

AND QUALIFIER
Specifies that the *role-name* will be used as the default for the CURRENT SCHEMA special register. The *role-name* will also be included in the SQL PATH (in place of CURRENT SQLID).

When **WITH ROLE AS OBJECT OWNER AND QUALIFIER** is not specified, there is no change to the default of the CURRENT SCHEMA special register and SQL PATH.

DISABLE or ENABLE
Specifies whether the trusted context is in the enabled or disabled state.

DISABLE
Specified that the trusted context is disabled. A trusted context that is disabled is not considered when a trusted connection is established.

ENABLE
Specifies that the trusted context is enabled.

NO DEFAULT SECURITY LABEL or DEFAULT SECURITY LABEL*seclabel-name*
Specifies whether a default security label is associated with a trusted connection that is based on this trusted context. If a trusted connection for the specified context is active, the change goes into effect at the next connection reuse attempt or when a new connection is requested.

NO DEFAULT SECURITY LABEL
Specifies that the trusted context does not have a default security label.

DEFAULT SECURITY LABEL *seclabel-name*
Specifies that *seclabel-name* is the default security label for the trusted context. *seclabel-name* is the security label that is used for multilevel security verification. *seclabel-name* must identify one of the RACF SECLABEL values that is defined for the SYSTEM AUTHID. This security label is used in a trusted connection that is based on the specified trusted context when the user does not have a specific security label defined as part of the definition of this trusted context. In this case, *seclabel-name* must also identify one of the RACF SECLABEL values that is defined for the user.

ALTER ATTRIBUTES or ADD ATTRIBUTES
Specifies a list of one or more connection trust attributes to change or add to the definition of a trusted context. The connection trust attributes are used to define the trusted context. If ALTER ATTRIBUTES is specified and the attribute is not currently part of the definition of the specified trusted context, an error is returned. Existing specifications for the specified attributes are changed to the new value if ALTER is specified. Attributes that are not specified retain the previously specified values.

ADDRESS *address-value*
Specifies the actual communication address that is used by the connection to communicate with the database manager. The protocol supported is only for TCP/IP. Previously specified ADDRESS values are removed when ALTER ATTRIBUTES is specified. The ADDRESS attribute can be specified multiple times, but each *address-value* must be unique.

When establishing a trusted connection, if multiple values are defined for the ADDRESS attribute for a trusted context, a candidate connection is considered to match this attribute if the address that is used by a connection matches any of the values that are defined for the ADDRESS attribute of the trusted context.

address-value specifies a string constant that contains the value that is associated with the ADDRESS trust attribute. *address-value* must be an IPv4 address, an IPv6 address, or a secure domain name with a length no greater than 254 bytes. No validation of *address-value* is done at the time the ALTER TRUSTED CONTEXT statement is processed. *address-value* must be left justified within the string constant.

- An IPv4 address is represented as a dotted decimal address. An example of an IPv4 address is 9.112.46.111.

- An IPv6 address is represented as a colon hexadecimal address. An example of an IPv6 address is 2001:0DB8:0000:0000:0008:0800:200C:417A. This address can also be express in a compressed form as 2001:DB8::8:800:200C:417A.
- A domain name is converted to an IP address by the domain name server where a resulting IPv4 or IPv6 address is determined. An example of a domain name is www.ibm.com. The gethostbyname socket call is used to resolve the domain name.

ENCRYPTION *encryption-value*
Specifies the minimum level of encryption of the data stream (network encryption) for the connection.

encryption-value specifies a string constant that contains the value that is associated with the ENCRYPTION trust attribute. *encryption-value* must be left justified within the string constant. ENCRYPTION must not be specified more than one time in the statement. *encryption-value* must be one of the following:

- NONE, which specifies that no specific level of encryption is required.
- LOW, which specifies that a minimum of light encryption is required. LOW corresponds to 64-bit DRDA encryption.
- HIGH, which specifies that strong encryption is required. HIGH corresponds to SSL encryption.

ENCRYPTION cannot be specified if ADD ATTRIBUTES is specified. See "CREATE TRUSTED CONTEXT" on page 1500 for more information about the ENCRYPTION attribute.

JOBNAME *jobname-value*
Specifies the z/OS job name or started task name (depending on the source of the address space) for local applications. Previously specified values for JOBNAME are removed when ALTER ATTRIBUTES is specified. The JOBNAME attribute can be specified multiple times, but each *jobname-value* must be unique.

jobname-value specifies a string constant that contains the value that is associated with the JOBNAME trust attribute. *jobname-value* is an EBCDIC 8 byte job name or started task name. *jobname-value* must be left justified within the string constant. The last character in the name can be a wildcard character (*) if the first character is an alphabetic character. If the job name ends with a wildcard, any job names that match the specified characters are considered for establishing the trusted connection.

The following table lists possible values for the job name depending on the source of the address space).

Table 108. Job name for local connection

Source of the address space	Job name
RRSAF	Job name or started task name
TSO	TSO logon ID
BATCH	Job name on JOB statement

SERVAUTH *servauth-value*
Specifies the name of a resource in the RACF SERVAUTH class. This resource is the network access security zone name that contains the IP address of the connection that is used to communicate with DB2. Previously specified values for SERVAUTH are removed when ALTER

ATTRIBUTES is specified. The SERVAUTH attribute can be specified multiple times but each *servauth-value* must be unique.

servauth-value specifies a string constant that contains the value that is associated with the SERVAUTH trust attribute. *servauth-value* is an EBCDIC 64 byte RACF SERVAUTH CLASS resource name. *servauth-value* must be left justified in the string constant. No validation of *servauth-value* is done at the time the ALTER TRUSTED CONTEXT statement is processed.

DROP ATTRIBUTES
Specifies that one or more attributes are dropped from the definition of a trusted context. If the attribute is not currently specified as part of the definition of a trusted context, an error is returned. The specification of DROP ATTRIBUTES must not attempt to drop all of the existing attributes for a trusted context.

ADDRESS *address-value*
Specifies that the identified communication address is removed from the definition of the trusted context. *address-value* specifies a string constant that contains the value of an existing ADDRESS trust attribute.

JOBNAME *jobname-value*
Specifies that the identified job name is removed from the definition of the trusted context. *jobname-value* specifies a string constant that contains the value of an existing JOBNAME trust attribute.

SERVAUTH *servauth-value*
Specifies that the identified servauth that is removed from the definition of the trusted context. *servauth-value* specifies a string constant that contains the value of an existing SERVAUTH trust attribute.

ADD USE FOR
Specifies additional users who can use a trusted connection that is based on the specified trusted context.

authorization-name
Specifies that the trusted connection can be used by the specified *authorization-name*. This is the DB2 primary authorization ID. The *authorization-name* must not identify an authorization ID that is already defined to use the trusted context, and must not be specified more than one time in the ADD USE FOR clause.

ROLE *role-name*
Specifies that *role-name* is the role that is used when a trusted connection is used by the specified *authorization-name*. The *role-name* must identify a role that exists at the current server. The role that is explicitly specified for the user overrides any default role that is associated with the trusted context.

SECURITY LABEL *seclabel-name*
Specifies that *seclabel-name* is the security label to use for multilevel security verification when the trusted connection is used by the specified *authorization-name*. The *seclabel-name* must be one of the RACF SECLABEL values that is defined for the user. The security label that is explicitly specified for the user overrides any default security label that is associated with the trusted context.

EXTERNAL SECURITY PROFILE *profile-name*
Specifies that the trusted connection can be used by the DB2 primary authorization IDs that are permitted to use the specified *profile-name* in RACF. The *profile-name* must not already be defined to use the trusted

context and must not be specified more than one time in the **ADD USE FOR** clause. After you specify an external security profile, any user who is permitted access to the RACF profile can use the trusted context in addition to any users that are specified using the **ADD USE FOR** *authorization-name* clause.

ROLE *role-name*
Specifies that *role-name* is the role that is used when a trusted connection is used by any authorization ID that is permitted to use the specified *profile-name* in RACF. The *role-name* must identify a role that exists at the current server. The role that is explicitly specified for the profile overrides any default role that is associated with the trusted context.

SECURITY LABEL *seclabel-name*
Specifies that *seclabel-name* is the security label to use for multilevel security verification when the trusted connection is used by any authorization ID that is permitted to use the specified *profile-name* in RACF. The *seclabel-name* must be one of the RACF SECLABEL values that is defined for the user. The security label that is explicitly specified for the profile overrides any default security label that is associated with the trusted context.

PUBLIC
Specifies that a trusted connection that is based on the specified trusted context can be used by any user. PUBLIC must not already be defined to use the trusted context and must not be specified more than one time in the **ADD USE FOR** clause.

All users that are using a trusted connection that is defined with PUBLIC use the privileges that are associated with the default role for the associated trusted context. If the default role is not defined for the trusted context, there is no role associated with the users that use a trusted connection that is based on the specified trusted context.

If the default security label for the trusted context is defined, all users that are using the trusted context must have the security label defined as one of the RACF SECLABEL values for the user. The default security label is used for multilevel security verification with all users that are using the trusted context.

The specifications for a user are determined in the following order of precedence:

- *authorization-name*
- EXTERNAL SECURITY PROFILE *profile-name*
- PUBLIC

For example, assume that a trusted context is defined with use for JOE WITH AUTHENTICATION, EXTERNAL SECURITY PROFILE SPROFILE WITHOUT AUTHENTICATION (with JOE and SAM permitted to use the RACF PROFILE SPROFILE), and PUBLIC WITH AUTHENTICATION. If the trusted connection is used by JOE, authentication is required. If the trusted connection is used by SAM, authentication is not required. However, if the trusted connection is used by SALLY, authentication is required.

REPLACE USE FOR
Specifies a change to the specified user or PUBLIC for who can use the trusted context.

authorization-name
Specifies the *authorization-name* that is changed for use of the trusted context. The trusted context must already be defined to allow use by *authorization-name*, and *authorization-name* must not be specified more than one time in the REPLACE USE FOR clause. The information that is associated with *authorization-name* is changed as indicated.

ROLE *role-name*
Specifies that *role-name* is the role that is used when a trusted connection is using the specified trusted context. The *role-name* must identify a role that exists at the current server. The role that is explicitly specified for the user overrides any default role that is associated with the trusted context.

SECURITY LABEL *seclabel-name*
Specifies that *seclabel-name* is the security label to use for multilevel security verification when the trusted connection is used by the specified *authorization-name*. The *seclabel-name* must be one of the RACF SECLABEL values that is defined for the user. The security label that is explicitly specified for the user overrides any default security label that is associated with the trusted context.

EXTERNAL SECURITY PROFILE *profile-name*
Specifies the *profile-name* to change attributes for use of the trusted connection. The trusted context must already be defined to allow the use of *profile-name*. *profile-name* must not be specified more than one time in the **REPLACE USE FOR** clause. The information that is associated with the profile name is changed as indicated.

ROLE *role-name*
Specifies that *role-name* is the role that is used when a trusted connection is used by any authorization ID that is permitted to use the specified *profile-name* in RACF. The role name must identify a role that exists at the current server. The role that is explicitly specified for the profile overrides any default role that is associated with the trusted context.

SECURITY LABEL *seclabel-name*
Specifies that *seclabel-name* is the security label to use for multilevel security verification when the trusted connection is used by any authorization ID that is permitted to use the specified *profile-name* in RACF. The *seclabel-name* must be one of the RACF SECLABEL values that is defined for the user. The security label that is explicitly specified for the user overrides any default security label that is associated with the trusted context.

PUBLIC
Specifies that the attributes for use of the trusted connection by PUBLIC are to be changed. PUBLIC must already be defined to use the trusted context, and PUBLIC must not be specified more than one time in the REPLACE USE FOR clause.

All users that are using a trusted connection that is defined with PUBLIC use the privileges that are associated with the default role for the associated trusted context. If the default role is not defined for the trusted context, there is no role associated with the users that use a trusted connection that is based on the specified trusted context.

If the default security label for the trusted context is defined, all users that are using the trusted context must have the security label defined as one of

the RACF SECLABEL values for the user. The default security label is used for multilevel security verification with all users that are using the trusted context.

WITHOUT AUTHENTICATION or WITH AUTHENTICATION
Specifies whether use of the trusted connection requires authentication of the user.

WITHOUT AUTHENTICATION
Specifies that use of a trusted connection by the user does not require authentication. WITHOUT AUTHENTICATION is the default.

WITH AUTHENTICATION
Specifies that use of a trusted connection requires the authentication token with the authorization ID to authenticate the user.

DROP USE FOR
Specifies who can no longer use the trusted context. The users that are removed from the definition of the trusted context are the specified users (or PUBLIC) that are currently allowed to use the trusted context. If multiple users are specified to be dropped, and one or more of those users cannot be dropped, those users that can be dropped are dropped and a warning is returned. If none of the specified users can be removed from the definition of the trusted context, an error is returned.

authorization-name
Specifies the *authorization-name* that will no longer be able to use this trusted context.

EXTERNAL SECURITY PROFILE *profile-name*
Removes the ability for the specified *profile-name* to use the trusted context.

PUBLIC
Specifies that PUBLIC users will no longer be able to use this trusted context. The system authorization ID and individual authorization IDs that have been explicitly enabled can still use the trusted context.

Notes

Precedence for** authorization-name **and authentication requirements: If the *authorization-name* that is specified in the SYSTEM AUTHID clause is the same authorization name that is specified in the ADD or REPLACE USE FOR *authorization-name* clauses, the role or the security label that is specified for the *authorization-name* takes precedence over the default value and the value that is specified for the EXTERNAL SECURITY PROFILE *profile-name* (if one is specified). If the authorization name that is specified in the SYSTEM AUTHID clause is permitted to use one of the specified profile names and is not specified in ADD or REPLACE USE for *authorization-name*, the role or the security label that is specified for that *profile-name* takes precedence over the default value.

Authentication is required for SYSTEM AUTHID if the AUTHENTICATION clause is specified in the ADD or REPLACE USE FOR clauses, or if the subsystem parameter TCP/IP Already Verified is set to NO. For example, if *authorization-name* is the same as the authorization name that is specified in the SYSTEM AUTHID clause and the WITHOUT AUTHENTICATION clause is specified, but the TCP/IP Already Verified subsystem parameter is set to NO, authentication is required for SYSTEM AUTHID when the remote trusted connection is established. If *authorization-name* is the SYSTEM AUTHID and the WITH AUTHENTICATION clause is specified, but the TCP/IP Already Verified subsystem parameter is set to

YES, authentication is still required for SYSTEM AUTHID.

Order of precedence for users of a trusted connection: The specifications for a user are determined in the following order of precedence:

- *authorization-name*
- **EXTERNAL SECURITY PROFILE** *profile-name*
- **PUBLIC**

For example, assume that a trusted context is defined with use for JOE WITH AUTHENTICATION, EXTERNAL SECURITY PROFILE SPROFILE WITHOUT AUTHENTICATION, and PUBLIC WITH AUTHENTICATION. Users JOE and SAM are permitted to use the RACF PROFILE SPROFILE. If the trusted connection is used by JOE, authentication is required. If the trusted connection is used by SAM, authentication is not required. However, if user SALLY uses the trusted connection, authentication is required.

User-clause SYSTEM AUTHID considerations: If the *authorization-name* that is specified in the SYSTEM AUTHID clause is the same as the *authorization-name* that is specified in the user-clause *authorization-name*, the role or the security label that is specified for *authorization-name* takes precedence over the default value. The value that is specified for the *profile-name*, is permitted to use the profile. If the authorization name that is specified in the SYSTEM AUTHID clause is permitted to use one of the profile names and is not defined in *authorization-name*, the role or the security label that is specified for that *profile-name* takes precedence over the default value.

If authentication is required for SYSTEM AUTHID, either by specification of the AUTHENTICATION clause in the *user-clause* or by setting the value of the TCP/IP Already Verified subsystem parameter to NO, the authentication requirement takes precedence when establishing a remote trusted connection. For example, if *authorization-name* is the same as the authorization name that is specified for SYSTEM AUTHID and the WITHOUT AUTHENTICATION clause is specified, but the TCP/IP Already Verified subsystem parameter is set to NO, an authentication token is required for SYSTEM AUTHID when the remote trusted connection is established. If *authorization-name* is the SYSTEM AUTHID and the WITH AUTHENTICATION clause is specified, but the TCP/IP Already Verified subsystem parameter is set to YES, an authentication token is still required for SYSTEM AUTHID.

Order of operations: The order in which the clauses of the ALTER TRUSTED CONTEXT statement are applied are as follows:

- DROP ATTRIBUTES
- DROP USE FOR
- ALTER
- ADD ATTRIBUTES
- ADD USE FOR
- REPLACE USE FOR

Effect of changes on existing trusted connections: If trusted connections exist for the trusted context that is changed, the connections continue to use the unchanged definition of the trusted context until the connection is terminated or an attempt at reuse is made. If the trusted context is disabled while there are active trusted connections that are based on this trusted context, the connections continue to be

used until terminated or an attempt at reuse is made. If the trust attributes are changed, trusted connections that exist at the time that the trusted context is changed will continue to be used.

When changes to a trusted context take place: The changes to the definition of a trusted context take effect after the ALTER TRUSTED CONTEXT statement is committed. If the ALTER TRUSTED CONTEXT statement results in an error or is rolled back, the trusted context is not changed.

Role privileges: If no role is associated with the user or the trusted context, only the privileges that are associated with the user are applicable. This is the same as not using a trusted context.

Examples

Example 1: The following statement updates the default role of the trusted context CTX1:

```
ALTER TRUSTED CONTEXT CTX1
    ALTER DEFAULT ROLE CTXROLE2;
```

Example 2: The following statement changes the CTX3 trusted context to allow use for BILL, and it also puts the trusted context into the disabled state:

```
ALTER TRUSTED CONTEXT CTX3
     DISABLE
     ADD USE FOR BILL;
```

Example 3: The following statement changes the CTX4 trusted context to allow the previously defined user JOE to use the trusted context without authentication. The statement also adds use for PUBLIC with authentication and TOM with a role of SPLROLE:

```
ALTER TRUSTED CONTEXT CTX4
   REPLACE USE FOR JOE WITHOUT AUTHENTICATION
   ADD USE FOR PUBLIC WITH AUTHENTICATION,
   TOM ROLE SPLROLE;
```

Example 4: The following statement changes the REMOTECTX to use a different IPv4 address than it was originally defined to use. It also changes the encryption settings from NONE to LOW. After the ALTER statement is processed, the connection will be considered trusted only when it is established from 9.12.155.200 with low encryption. The connection will no longer be considered trusted if it is established from the previously defined addresses:

```
ALTER TRUSTED CONTEXT REMOTECTX
   ALTER ATTRIBUTES (ADDRESS '9.12.155.200',
                     ENCRYPTION 'LOW');
```

ALTER VIEW

The ALTER VIEW statement regenerates a view using an existing view definition at the current server. ALTER VIEW is primarily used during DB2 migration or when DB2 maintenance is applied. To change a view definition (for example, to add additional columns), you must drop the view and create a new view using the CREATE VIEW statement.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

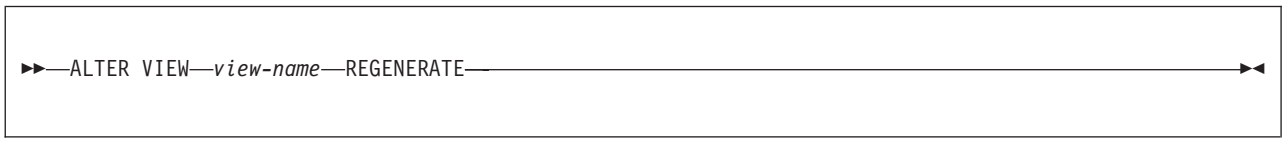
Authorization

The privilege set that is defined below must include at least one of the following:

- Ownership of the view
- SYSADM authority
- SYSCTRL authority
- System DBADM

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the package. If the statement is dynamically prepared, the privilege set is the union of the privilege sets that are held by each authorization ID and role of the process.

Syntax



Description

view-name
Identifies the view to be regenerated. The name must identify a view that exists at the current server.

REGENERATE
Specifies that the view is to be regenerated. The view definition in the catalog is used, and existing authorizations and dependent views are retained. The catalog is updated with the regenerated view definition. If the view cannot be successfully regenerated, an error is returned.

Examples

Check the catalog to find any views that were marked with view regeneration errors during catalog migration:

```
SELECT CREATOR,NAME FROM SYSIBM.SYSTABLES
  WHERE TYPE = 'V' AND STATUS = 'R' AND TABLESTATUS = 'V';
```

Assume that the query returned MYVIEW as the name of a view with a regeneration error. Issue an ALTER VIEW statement to regenerate the view:

```
ALTER VIEW MYVIEW REGENERATE;
```

ASSOCIATE LOCATORS

The ASSOCIATE LOCATORS statement gets the result set locator value for each result set returned by a stored procedure.

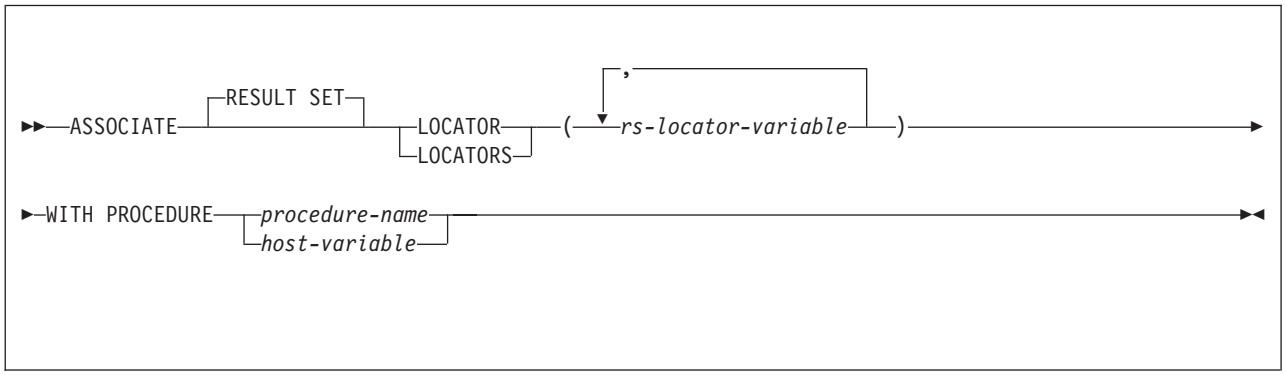
Invocation

This statement can be embedded in an application program. It is an executable statement that can be dynamically prepared. It cannot be issued interactively.

Authorization

None required.

Syntax



Description

rs-locator-variable
Identifies a result set locator variable that has been declared according to the rules for declaring result set locator variables.

WITH PROCEDURE *procedure-name* **or** *host-variable*
Identifies the stored procedure that returned one or more result sets. When the ASSOCIATE LOCATORS statement is executed, the procedure name must identify a stored procedure that the requester has already invoked using the SQL CALL statement. The procedure name can be specified as a one-part, two-part, or three-part name. The procedure name in the ASSOCIATE LOCATORS statement must be specified the same way that it was specified on the CALL statement. For example, if a two-part procedure name was specified on the CALL statement, you must specify a two-part procedure name in the ASSOCIATE LOCATORS statement.

If a host variable is used to specify the name:

- It must be a character string variable with a length attribute that is not greater than 255.
- It must not be followed by an indicator variable.
- The value of the host variable is a specification that depends on the server. Regardless of the server, the specification must:
  - Be left justified within the host variable
  - Not contain embedded blanks
  - Be padded on the right with blanks if its length is less than that of the host variable

Notes

Assignment of locator values: If the ASSOCIATE LOCATORS statement specifies multiple locator variables, locator values are assigned to the locator variables in the order that the associated cursors are opened regardless of whether they are still open or not at run time. Locator values are assigned to the locator variables in the same order that they would be placed in the SQLVAR entries in the SQLDA as a result of a DESCRIBE PROCEDURE statement.

Locator values are not provided for cursors that are closed when control is returned to the invoking application. If a cursor was closed and later opened again before returning to the invoking application, the most recently executed OPEN CURSOR statement for the cursor is used to determine the order in which the locator values are returned for the procedure result sets. For example, assume procedure P1 opens three cursors A, B, C, closes cursor B and then issues another OPEN CURSOR statement for cursor B before returning to the invoking application. The locator values assigned for the following ASSOCIATE LOCATORS statement will be in the order A, C, B:

```
ASSOCIATE RESULT SET LOCATORS (:loc1, :loc2, :loc3) WITH PROCEDURE P1;
   -- assigns locators for result set cursors A, C, and B
```

More than one locator can be associated with a result set. You can issue multiple ASSOCIATE LOCATORS statements for the same stored procedure with different result set locator variables to associate multiple locators with each result set.

- If the number of result set locator variables specified in the ASSOCIATE LOCATORS statement is less than the number of result sets returned by the stored procedure, all locator variables specified in the statement are assigned a value, and a warning is issued. For example, assume procedure P1 exists and returns four result sets. Each of the following ASSOCIATE LOCATORS statement returns information on the first result set along with a warning that not enough locators were provided to obtain information about all the result sets.

  ```
  CALL P1;
  ASSOCIATE RESULT SET LOCATORS (:loc1) WITH PROCEDURE P1;
    -- :loc1 is assigned a value for first result set, and a warning is returned
  ASSOCIATE RESULT SET LOCATORS (:loc2) WITH PROCEDURE P1;
       -- :loc2 is assigned a value for first result set, and a warning is returned
  ASSOCIATE RESULT SET LOCATORS (:loc3) WITH PROCEDURE P1;
       -- :loc3 is assigned a value for first result set, and a warning is returned
  ASSOCIATE RESULT SET LOCATORS (:loc4) WITH PROCEDURE P1;
       -- :loc4 is assigned a value for first result set, and a warning is returned
  ```

- If the number of result set locator variables that are listed in the ASSOCIATE LOCATORS statement is greater than the number of locators returned by the stored procedure, the extra locator variables are assigned a value of 0.

Accessing result sets from multiple CALL statements: An application can access to result sets created by multiple CALL statements. The result sets can be created by different procedure or by the same procedure invoked multiple times.

- *Invoking different procedures:* Invoking different procedures with the same name can be done either explicitly by specifying the different collections or implicitly with the use of the PACKAGE PATH. For example, to identify the different collections explicitly, specify qualified names on the CALL statement. Although both procedures are named P2, they are different procedures. After the second CALL statement, result sets from both procedures are accessible to the application.

```
CALL X.P2;
CALL Y.P2;
```

The collections for the two different procedures can also be determined implicitly from the PACKAGE PATH when unqualified procedure names are specified as part of the CALL statement. For example, assume that procedure P4 exists in collections X and Z. An application contains two CALL statements to invoke procedure P4. The references to procedure P4 in the CALL statements are unqualified. So, the PACKAGE PATH special register is used to resolve the procedure name. Procedure X.P4 is invoked for the first CALL statement and procedure Z.P4 is invoked by the second CALL statement. Following the second CALL statement, result sets from both procedures are accessible to the application.

```
SET CURRENT PACKAGE PATH = X, Y, Z;
CALL P4;
SET CURRENT PACKAGE PATH = PATH Z, Y, X;
CALL P4;
```

- *Invoking the same procedure multiple times:* If the server and requester are both the same version of DB2, you can call a stored procedure multiple times within an application and at the same nesting level. Each call to the same stored procedure causes a unique instance of the stored procedure to run. If the stored procedure returns result sets, each instance of the stored procedure opens its own set of result set cursors. For more information on this situation, see Multiple calls to the same stored procedure.

  When a procedure is invoked multiple times in an application and there is a need to process the result sets from the different instances at the same time, be sure to use the ASSOCIATE LOCATORS statement after each CALL statement to capture the locator values returned from each invocation of the procedure. For example, assume that procedure P exists in collection Z and that an application contains two CALL statements to invoke procedure P. The PACKAGE PATH is used to determine the collection for the procedure in the first CALL statement, and the collection is explicitly specified in the second CALL statement. Result sets from both procedures can be accessible to the application following both CALL statements if the locators for the result sets produced by the first CALL statement are captured with an ASSOCIATE LOCATOR statement before invoking the procedure the second time.

  ```
  SET CURRENT PACKAGE PATH = X, Y, Z;
  CALL P3;
  ASSOCIATE LOCATORS ...
  CALL Z.P3;
  ASSOCIATE LOCATORS ...
  -- process the result sets using the locators
  ```

Using host variables: If the ASSOCIATE LOCATORS statement contains host variables, the following conditions apply:

- If the statement is executed statically, the contents of the host variables are assumed to be in the encoding scheme that was specified in the ENCODING parameter when the package or plan that contains the statement was bound.
- If the statement is executed dynamically, the contents of the host variables are assumed to be in the encoding scheme that is specified in the APPLICATION ENCODING bind option.

Examples

The statements in the following examples are assumed to be in PL/I programs.

Example 1: Use result set locator variables LOC1 and LOC2 to get the result set locator values for the two result sets returned by stored procedure P1. Assume that the stored procedure is called with a one-part name from current server SITE2.

```
EXEC SQL CONNECT TO SITE2;
EXEC SQL CALL P1;
EXEC SQL ASSOCIATE RESULT SET LOCATORS (:LOC1, :LOC2)
         WITH PROCEDURE P1;
```

Example 2: Repeat the scenario in Example 1, but use a two-part name to specify an explicit schema name for the stored procedure to ensure that stored procedure P1 in schema MYSCHEMA is used.

```
EXEC SQL CONNECT TO SITE2;
EXEC SQL CALL MYSCHEMA.P1;
EXEC SQL ASSOCIATE RESULT SET LOCATORS (:LOC1, :LOC2)
         WITH PROCEDURE MYSCHEMA.P1;
```

Example 3: Use result set locator variables LOC1 and LOC2 to get the result set locator values for the two result sets that are returned by the stored procedure named by host variable HV1. Assume that host variable HV1 contains the value SITE2.MYSCHEMA.P1 and the stored procedure is called with a three-part name.

```
EXEC SQL CALL SITE2.MYSCHEMA.P1;
EXEC SQL ASSOCIATE LOCATORS (:LOC1, :LOC2)
         WITH PROCEDURE :HV1;
```

The preceding example would be invalid if host variable HV1 had contained the value MYSCHEMA.P1, a two-part name. For the example to be valid with that two-part name in host variable HV1, the current server must be the same as the location name that is specified on the CALL statement as the following statements demonstrate. This is the only condition under which the names do not have to be specified the same way and a three-part name on the CALL statement can be used with a two-part name on the ASSOCIATE LOCATORS statement.

```
EXEC SQL CONNECT TO SITE2;
EXEC SQL CALL SITE2.MYSCHEMA.P1;
EXEC SQL ASSOCIATE LOCATORS (:LOC1, :LOC2)
         WITH PROCEDURE :HV1;
```

BEGIN DECLARE SECTION

The BEGIN DECLARE SECTION statement marks the beginning of an SQL declare section. An SQL declare section contains declarations of host variables that are eligible to be used as host variables in SQL statements in a program.

Invocation

This statement can only be embedded in an application program. It is not an executable statement. It must not be specified in Java or REXX.

Authorization

None required.

Syntax

Description

The BEGIN DECLARE SECTION statement can be coded in the application program wherever variable declarations can appear in accordance with the rules of the host language. It is used to indicate the beginning of a host variable declaration section. A host variable section ends with an END DECLARE SECTION statement, described in "END DECLARE SECTION" on page 1631.

The following rules are enforced by the precompiler only if the host language is C or the STDSQL(YES) SQL processing option is specified:

- A variable referred to in an SQL statement must be declared within a host variable declaration section of the source program in all host languages, other than Java and REXX. Furthermore, the declaration of each variable must appear before the first reference to the variable. Host variables are declared without the use of these statements in Java, and they are not declared at all in REXX.
- BEGIN DECLARE SECTION and END DECLARE SECTION statements must be paired and must not be nested.
- Host variable declaration sections can contain only host variable declarations, SQL INCLUDE statements that include host variable declarations, or DECLARE VARIABLE statements.

Notes

Host variable declaration sections are only required if the STDSQL(YES) option is specified or the host language is C. However, declare sections can be specified for any host language so that the source program can conform to IBM SQL. If declare sections are used, but not required, variables declared outside a declare section must not have the same name as variables declared within a declare section.

Example

```
EXEC SQL BEGIN DECLARE SECTION;

 -- host variable declarations

EXEC SQL END DECLARE SECTION;
```

CALL

The CALL statement invokes a stored procedure.

Invocation

This statement can be embedded in an application program. This statement can be executed interactively using the command line processor. Refer to *DB2 Application Programming and SQL Guide* for information about using the command line processor with the CALL statement. This statement can also be dynamically prepared, but only from an ODBC or CLI driver that supports dynamic CALL statements. IBM's ODBC and CLI drivers provide this capability.

Authorization

Invoking a stored procedure requires the EXECUTE privilege on the following:

- The stored procedure

  You do not need the EXECUTE privilege on a stored procedure that was created prior to Version 6 of DB2 for z/OS.
- For external procedures (including external SQL procedures), additional authority is needed for the stored procedure package and most packages that run in the stored procedure.

  The authorization that is required for which packages is explained in detail in Authorization to execute packages under the stored procedure.

Authorization to execute the stored procedure

The authorization ID or role that must have the EXECUTE privilege on the stored procedure depends on the form of the CALL statement:

- For static SQL programs that use the syntax CALL *procedure*, the owner of the plan or package that contains the CALL statement must have one of the following:
  - The EXECUTE privilege on the stored procedure
  - Ownership of the stored procedure
  - DATAACCESS authority
  - SYSADM authority
- For static SQL programs that use the syntax CALL *:host-variable*, the authorization ID or role of the plan or package that contains the CALL statement must have one of the following:
  - The EXECUTE privilege on the stored procedure
  - Ownership of the stored procedure
  - DATAACCESS authority
  - SYSADM authority

  The DYNAMICRULES behavior for the plan or package that contains the CALL statement determines both the authorization ID or role and the privilege set that is held by that authorization ID or role:

  Run behavior
  : The privilege set is the union of the set of privileges that are held by the SQL authorization ID and each authorization ID or role of the process.

Bind behavior
The privilege set is the privileges that are held by the primary authorization ID of the owner of the package or plan.

Define behavior
The privilege set is the privileges that are held by the authorization ID or role of the owner (definer) of the stored procedure or user-defined function that issued the CALL statement.

Invoke behavior
The privilege set is the privileges that are held by the authorization ID or role of the invoker of the stored procedure or user-defined function that issued the CALL statement. However, if the invoker is the primary authorization ID of the process or the CURRENT SQLID value, the privilege set is the union of the set of privileges that are held by each authorization ID or role.

For a list of the DYNAMICRULES values that specify run, bind, define, or invoke behavior, see Table 6 on page 75.

Authorization to execute packages under the stored procedure (including nested stored procedures)

The authorization that is required to run the stored procedure package and any packages that are used under the stored procedure (including nested stored procedures) apply to any form of the CALL statement as follows:

- ***Stored procedure package:*** One of the authorization IDs or roles that are defined in Set of authorization IDs must have at least one of the following privileges or authorities on the stored procedure package:
  - The EXECUTE privilege
  - Ownership of the package
  - PACKADM authority for the package's collection
  - SYSADM authority

  A PKLIST entry is not required for the stored procedure package.
- ***User-defined function packages and trigger packages:*** If a stored procedure or any application under the stored procedure invokes a user-defined function, DB2 requires only the owner (the definer), and not the invoker of the user-defined function, to have EXECUTE authority on the user-defined function package. However, the authorization ID or role of the SQL statement that invokes the user-defined function must have EXECUTE authority on the function.

  Similarly, if a trigger is used under a stored procedure, DB2 does not require EXECUTE authority on the trigger package; however, the authorization ID or role of the SQL statement that activates the trigger must have EXECUTE authority on the trigger.

  For more information about the EXECUTE authority for user-defined functions, triggers, and user-defined function packages, see *DB2 Administration Guide*.

  PKLIST entries are not required for any user-defined function packages or trigger packages that are used under the stored procedure.
- ***Packages other than user-defined function, trigger, and stored procedure packages:*** One of the authorization IDs or roles that is defined below under Set of authorization IDs must have at least one of the following privileges or authorities on any packages other than user-defined function and trigger packages that are used under the stored procedure:
  - The EXECUTE privilege

- Ownership of the package
- PACKADM authority for the package's collection
- SYSADM authority

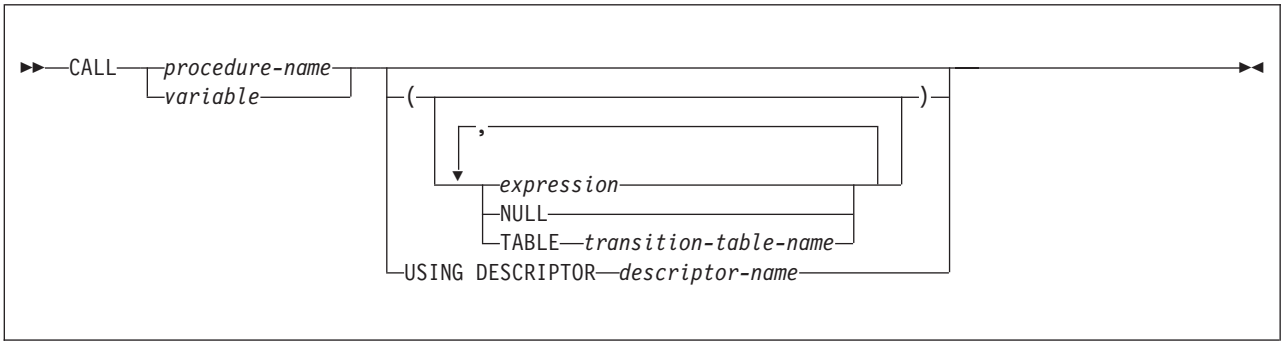
PKLIST entries are required for any of these packages that are used under the stored procedure.

For improved performance and simplicity, consider granting the EXECUTE ON PACKAGE privilege for the stored procedure package, and for any packages that run under the stored procedure, to the owner of the stored procedure.

Set of authorization IDs: DB2 checks the following authorization IDs, in the order in which they are listed, for the required authorization to execute the stored procedure package and any packages that run under the stored procedure other than user-defined function and trigger packages as described previously. Authorization checking ends after the first authorization ID that has EXECUTE ON PACKAGE privileges for the target package is found.

- The owner (the definer) of the stored procedure.
- The owner of the plan that contains the CALL statement that invokes the stored procedure if either of the following conditions is true:
  - The calling application (a package or a DBRM that is bound directly to the plan) is local.
  - The calling application is distributed, the DB2 subsystem is both the requester and the server, and the PRIVATE_PROTOCOL subsystem parameter is not set to NO.
- The owner of the package that contains the CALL statement that invokes the stored procedure if the calling application is distributed and either of the following conditions is true:
  - The DB2 subsystem is the server but not the requester.
  - The DB2 subsystem is both the server and the requester and the PRIVATE_PROTOCOL subsystem parameter is set to NO.
  - The calling application uses Recoverable Resources Management Services attachment facility (RRSAF) and has no plan.
- The authorization ID as determined by the value of the DYNAMICRULES bind option for the plan or package that contains the CALL statement if the CALL statement is in the form of CALL *:host-variable*.
  - If the calling application is bound with the DYNAMICRULES(RUN) option, DB2 checks either the authorization ID of the process at run time and its secondary authorization IDs or the single authorization ID that is determined by the other DYNAMICRULES bind option values.
  - If the calling application is bound with a value other than DYNAMICRULES(RUN), DB2 checks only a single authorization ID, even if that ID fails the EXECUTE ON PACKAGE authorization check.
  - If the calling application is a package and is bound with DYNAMICRULES(BIND), DB2 checks the authorization ID of the package owner. DB2 does not check the authorization ID of the plan owner.

Syntax



Description

procedure-name **or** *host-variable*
Identifies the procedure to call by the specified *procedure-name* or the procedure name contained in the *host-variable*. The identified procedure must exist at the current server.

If *procedure-name* specifies any of the three special characters that are alphabetic extenders for national languages, $#@, specify the procedure name with a *host-variable*.

If a host variable is used:

- It must be a CHAR or VARCHAR variable with a length attribute that is not greater than 254.
- It must not be followed by an indicator variable.
- The value of the host variable is a specification that depends on the server. Regardless of the server, the specification must:
  - Be left justified within the host variable
  - Not contain embedded blanks
  - Be padded on the right with blanks if its length is less than that of the host variable

  In addition, the specification can:
  - Contain upper and lowercase characters. Lowercase characters are not folded to uppercase.
  - Use a delimited identifier for any part of a three-part procedure name.

  If the server is DB2 for z/OS, the specification must be a procedure name as defined above.

When the CALL statement is executed, the procedure name or specification must identify a stored procedure that exists at the server.

When the package that contains the CALL statement is bound, the stored procedure that is invoked must be created if VALIDATE(BIND) is specified. Although the stored procedure does not need to be created at bind time if VALIDATE(RUN) is specified, it must be created when the CALL statement is executed.

expression, **NULL, or TABLE** *transition-table-name*
Identifies a list of values to be passed as arguments to the stored procedure. The *n*th value corresponds to the *n*th parameter in the procedure. Each parameter that is defined using CREATE PROCEDURE as OUT or INOUT

must be specified as a variable. An argument that is an array can be specified only if the CALL statement is issued from SQL PL.

The number of arguments that are specified must be the same as the number of parameters of a procedure that is defined at the current server with the specified procedure name.

If USING DESCRIPTOR is specified, each host variable described by the identified SQLDA is an argument, or part of an expression that is an argument of the CALL statement. If host structures are not specified in the CALL statement, the *n*th argument of the CALL statement corresponds to the *n*th parameter in the stored procedure, and the number in each must be the same. Otherwise, each reference to a host structure is replaced by a reference to each of the variables contained in that host structure, and the resulting number of arguments must be the same as the number of parameters defined for the stored procedure.

However, a character FOR BIT DATA argument cannot be passed as input for a parameter that is not defined as character FOR BIT DATA. Likewise, a character argument that is not FOR BIT DATA cannot be passed as input for a parameter that is defined as character FOR BIT DATA.

The attributes of the parameters are determined by the current server. In addition to attributes such as data type and length, the description of each parameter indicates how the stored procedure uses it:

- IN means as an input value
- OUT means as an output value
- INOUT means both as an input and an output value

All parameters that are not variables are assumed to be input parameters (IN).

expression
The argument is the result of the specified expression, which is evaluated before the stored procedure is invoked.

If *expression* is a single variable, the corresponding parameter of the procedure can be defined as IN, INOUT, or OUT. Otherwise, the corresponding parameter of the procedure must be defined as IN. An expression can contain any of the following items:
- Variable
- Constant
- Special register
- Session global variable
- Cast function with a variable or constant argument

A variable can identify a structure. Any variable or structure that is specified must be described in the application program according to the rules for declaring host structures and variables. A reference to a host structure is replaced by a reference to each of the variables contained in the host structure.

If the result of the expression can be the null value, either the description of the procedure must allow for null parameters or the corresponding parameter of the stored procedure must be defined as OUT.

expression must not reference an associative array value as an argument to a function if the procedure is remote.

The following additional rules apply depending on how the corresponding parameter was defined in the CREATE PROCEDURE statement for the procedure:

- IN *expression* can contain references to multiple host variables. In addition to the rules stated in "Expressions" on page 240 for *expression*, *expression* cannot include a column name, a scalar subselect, a file reference variable, an aggregate function, or a user-defined function that is sourced on an aggregate function.
- INOUT or OUT *expression* can only be a single variable. *expression* cannot include a file reference variable or an array element.

NULL
The parameter is a null value. The corresponding parameter of the procedure must be defined as IN and the description of the procedure must allow for null parameters.

TABLE *transition-table-name*
The parameter is a transition table, and it is passed to the procedure as a table locator. You can use the CALL statement with the TABLE clause only within the definition of the triggered action of a trigger. The name of a transition table must be specified in the CALL statement if the corresponding parameter of the procedure was defined in the TABLE LIKE clause of the CREATE PROCEDURE statement. For information about creating a trigger, see "CREATE TRIGGER" on page 1482 and *DB2 Application Programming and SQL Guide*.

There is no effect on the transition table on the return from the procedure regardless of whether the parameter was defined as IN, INOUT, or OUT.

USING DESCRIPTOR *descriptor-name*
Identifies an SQLDA that contains a valid description of the host variables that are to be passed as parameters to the stored procedure. If the stored procedure has no parameters, an SQLDA is ignored.

Before the CALL statement is processed, the user must set the following fields in the SQLDA:

- SQLN to indicate the number of SQLVAR occurrences provided in the SQLDA. This number must not be less than SQLD. This field is not part of the REXX SQLDA and therefore does not need to be set for REXX programs.
- SQLDABC to indicate the number of bytes of storage allocated for the SQLDA. This number must be not be less than SQLN*44+16. This field is not part of the REXX SQLDA and therefore does not need to be set for REXX programs.
- SQLD to indicate the number of variables used in the SQLDA when processing the statement. This number must be the same as the number of parameters of the stored procedure.
- SQLVAR occurrences to indicate the attributes of the variables.

There are additional considerations for setting the fields of the SQLDA when a variable that is passed as a parameter to the stored procedure has a LOB data type or is a LOB locator. For more information, see "SQL descriptor area (SQLDA)" on page 2079.

The SQL CALL statement ignores distinct type information in the SQLDA. Only the base SQL type information is used to process the input and output parameters described by the SQLDA.

In REXX, only *host variables* USING DESCRIPTOR is supported. Since global variables are not supported within the SQLDA, global variable are not supported in REXX.

See "Identifying an SQLDA in C or C++" on page 2099 for how to represent *descriptor-name* in C.

Notes

Parameter assignments: When the CALL statement is executed, the value of each of its arguments is assigned with storage assignment rules to the corresponding IN or INOUT parameter of the stored procedure. In cases where the arguments of the CALL statement are not an exact match to the data types of the parameters of the stored procedure, each argument specified in the CALL statement is converted to the data type of the corresponding parameter of the stored procedure at execution. The conversion occurs according to the same rules as assignment to columns.

Control is passed to the stored procedure according to the calling conventions of the host language.

When execution of the stored procedure is complete, the value of each parameter of the stored procedure defined as OUT or INOUT is assigned to the corresponding argument of the CALL statement. If an error is returned by the procedure, OUT arguments are undefined, and INOUT arguments are unchanged.

A timestamp without time zone value must not be assigned to a timestamp with time zone target.

The following rules apply when the value of an array argument is assigned to the corresponding array parameter:

- **For a local procedure call:** The argument and the parameter must be defined as the same array type.
- **For a remote procedure call:** The data type of the elements of the array argument must be the same as the data type of the elements of the array parameter. In addition, for IN or OUT parameters, all of the relevant conditions in one of the rows in the following table must be true. For INOUT parameters, all of the relevant conditions in row 1 of the following table must be true, or all of the relevant conditions in rows 2 and 3 must be true. A relevant condition is indicated with Y.

Relationship of argument to associated parameter	Relationship applies to IN parameter	Relationship applies to OUT parameter	Relationship applies to INOUT parameter
The argument is an ordinary array, the parameter is an ordinary array, and the argument and parameter are defined with the same data type for the array indexes.	Y	Y	Y

Relationship of argument to associated parameter	Relationship applies to IN parameter	Relationship applies to OUT parameter	Relationship applies to INOUT parameter
The argument is an ordinary array, the parameter is an associative array type, the parameter is an IN or INOUT parameter, and the data type of the array indexes is INTEGER. The associative array parameter is assigned an associative array value that is derived from the ordinary array argument value. The values of the array elements in the ordinary array are assigned to the target associative array parameter, in the same order as their order in the ordinary array argument. The index values in the target associative array parameter are assigned from 1 to the cardinality of the ordinary array argument value.	Y		Y
The argument is an ordinary array type, the parameter is an associative array type, and the parameter is an INOUT or OUT parameter. The argument is assigned an ordinary array value that is derived from the associative array parameter value. The values of the array elements in the associative array value are assigned to the target ordinary array, in an order determined by DB2. The index values in the target ordinary array argument are assigned from 1 to the cardinality of the associative array parameter value. The index values from the associative array parameter value are ignored.		Y	Y

For details on the rules used to assign parameters, see "Assignment and comparison" on page 121.

Conversion can occur when precision, scale, length, encoding scheme, or CCSID differ between the argument specified in the CALL statement and the data type of the corresponding parameter of the stored procedure. Conversion might occur for a character string argument specified in the CALL statement when the corresponding parameter of the stored procedure has a different encoding scheme or CCSID. For example, an error occurs when the CALL statement passes an argument of mixed data that actually contains DBCS characters as input for a parameter of the stored procedure that is defined as FOR SBCS DATA. Likewise, an error occurs when the

stored procedure returns mixed data that actually contains DBCS characters for an argument of the CALL statement that is defined as FOR SBCS DATA.

Procedure signatures: A procedure is identified by its schema, a procedure name, and its number of parameters. This is called a procedure signature, which must be unique within the database. DB2 for z/OS does not support overloaded procedure names (procedures with the same schema and procedure name, but with different numbers of parameters).

SQL path: A procedure can be invoked by referring to a qualified name (schema and procedure name), followed by an optional list of arguments that are enclosed in parentheses. A procedure can also be invoked without the schema name, which results in a choice of possible procedures in different schemas that have the same procedure name and same number of parameters. In this case, the SQL path is used to assist in procedure resolution. The SQL path is a list of schemas that is searched to identify a procedure with the same name and number of parameters as the procedure in the CALL statement. For CALL statements that explicitly specify a procedure name, the SQL path is specified by using the platform-specific bind option. For CALL *host-variable* statements, the SQL path is the value of the CURRENT PATH special register when the procedure is invoked.

Procedure resolution: Given a procedure invocation, the database manager must decide which of the possible procedures that has the same name to call.

A procedure name is a qualified or unqualified name. Each part of the name must be composed of SBCS characters:

- A fully qualified procedure name is a three-part name. The first part is an SQL identifier that contains the location name that identifies the DBMS at which the procedure is stored. The second part is an SQL identifier that contains the schema name of the stored procedure. The last part is an SQL identifier that contains the name of the stored procedure. A period must separate each of the parts. Any or all of the parts can be a delimited identifier.
- A two-part procedure name has one implicit qualifier. The implicit qualifier is the location name of the current server. The two parts identify the schema name and the name of the stored procedure. A period must separate the two parts.
- An unqualified procedure name is a one-part name with two implicit qualifiers. The first implicit qualifier is the location name of the current server. The second implicit qualifier depends on the server. If the server is DB2 for z/OS, the implicit qualifier is the schema name. DB2 uses the SQL path to determine the value of the schema name.
  - If the procedure name is specified as a string constant on the CALL statement (CALL *procedure-name*), the SQL path is the value of the PATH bind option that is associated with the calling package or plan.
  - If a host variable is specified for the procedure name on the CALL statement (CALL *host-variable*), the SQL path is the value of the CURRENT PATH special register.

  DB2 searches the schema names in the SQL path from left to right until a stored procedure with the specified schema name is found in the DB2 catalog. When a matching *schema.procedure-name* is found, the search stops only if the following conditions are true:
  - The user is authorized to call the stored procedure.
  - The number of parameters in the definition of the stored procedure matches the number of parameters specified on the CALL statement.

If the list of schemas in the SQL path is exhausted before the procedure name is resolved, an error is returned.

When the procedure is resolved depends on how the procedure name is specified. For a CALL statement that specifies the procedure name using a host variable, procedure resolution occurs at run time. For a CALL statement that contains the name of the procedure as an identifier, procedure resolution occurs when the CALL statement is bound.

Procedure resolution is done by the database manager using the following steps:

1. Find all procedures from the catalog where all of the following conditions are true:
   - For invocations where the schema name is specified (qualified references), the schema name and the procedure name match the invocation name.

     For invocations where the schema name is not specified (unqualified references), the procedure name matches the invocation name, and the procedure has a schema name that matches one of the schemas in the SQL path.
   - The number of defined parameters matches the number of arguments that are specified in the invocation.
   - The invoker has the EXECUTE privilege on the procedure.
2. Of the candidate procedures that remain from step 1, choose the procedure whose schema is first in the SQL path. If no candidate procedures remain after step 1, an error is returned.
3. For CALL statements that use a host variable to specify the procedure name, the CURRENT ROUTINE VERSION special register can affect which version of the native SQL procedure is invoked. If the CURRENT ROUTINE VERSION special register is set, check if there is a version of the procedure with that version name. If not, choose the currently active version of the procedure.

   For CALL statements that do not use a host variable to specify the procedure name, choose the currently active version of the procedure.

Version resolution: Normally, the currently active version of a native SQL procedure will be used on a CALL statement. However, if the CALL statement is a recursive call inside the body of the same stored procedure, and the original CALL statement uses a version that is different from the currently active version, the active version will not be used. The version from the original CALL statement will be used for any recursive CALL statements until the entire stored procedure finishes executing. This preserves the semantics of the version that is used by the original CALL statement. This includes the case where the recursive call is indirect. For example, assume that procedure SP1 call procedure SP2, which in turn recursively calls SP1. The second invocation of procedure SP1 will use the version of the procedure that is active at the time of the original CALL statement that invoked procedure SP1.

Since the currently active version can be used at the next CALL statement, it is possible that two or more versions of the same procedure can run at the same time. There could be different versions of an SQL procedure loaded by a given thread. For example, a CALL SP1 statement in an application will cause the currently active version, SP1_V1, to load and execute. After this CALL statement has completed, an ALTER PROCEDURE ALTER ACTIVE VERSION could execute and change the active version of the procedure SP1 to version SP1_V2. Subsequent CALL SP1 statements from the same thread will load the currently active version of the procedure, SP1_V2, and execute it.

Parameter assignments: When the CALL statement is executed, the value of each of its parameters is assigned with storage assignment rules to the corresponding parameter of the procedure. Control is passed to the procedure according to the calling conventions of the host language. When execution of the procedure is complete, the value of each parameter of the procedure is assigned with storage assignment rules to the corresponding parameter of the CALL statement defined as OUT or INOUT. If an error is returned by the procedure, OUT arguments are undefined and INOUT arguments are unchanged. For details on the assignment rules, see "Assignment and comparison" on page 121.

Cursors and prepared statements in procedures: All cursors opened in the called procedure that are not result set cursors are closed and all statements prepared in the called procedure are destroyed when the procedure ends.

Result sets from procedures: Any cursors specified using the WITH RETURN clause that the procedure leaves open when it returns identifies a result set. In a procedure written in Java, all cursors are implicitly defined WITH RETURN.

Results sets are returned only when the procedure is called from CLI, JDBC, or SQLJ. If the procedure was invoked from CLI or Java, and more than one cursor is left open, the result sets can only be processed in the order in which the cursors were opened. Only unread rows are available to be fetched. For example, if the result set of a cursor has 500 rows, and 150 of those rows have been read by the procedure at the time the procedure is terminated, then rows 151 through 500 will be returned to the procedure.

Errors from procedures: A procedure can return errors or warnings using an SQLSTATE-like SQL statement. Applications should be aware of the possible SQLSTATEs that can be expected when a procedure is invoked. The possible SQLSTATEs depend on how the procedure is coded. Procedures might also return SQLSTATEs such as those that begin with '38' or '39' if DB2 encounters problems executing the procedure. Applications should therefore be prepared to handle any error SQLSTATE that can result from issuing a CALL statement.

Improving performance: The capability of calling stored procedures is provided to improve the performance of DRDA distributed access. The capability is also useful for local operations. The server can be the local DB2. In which case, packages are still required.

All values of all parameters are passed from the requester to the server. To improve the performance of this operation, host variables that correspond to OUT parameters and have lengths of more than a few bytes should be set to null before the CALL statement is executed.

Using the CALL statement in a trigger: When a trigger issues a CALL statement to invoke a stored procedure, the parameters that are specified in the CALL statement cannot be host variables and the USING DESCRIPTOR clause cannot be specified.

Nesting CALL statements: A program that is executing as a stored procedure, a user-defined function, or a trigger can issue a CALL statement. When a stored procedure, user-defined function, or trigger calls a stored procedure, user-defined function, or trigger, the call is considered to be nested. Stored procedures, user-defined functions, and triggers can be nested up to 64 levels deep on a single system. Nesting can occur within a single DB2 subsystem or when a stored procedure or user-defined function is invoked at a remote server.

If a stored procedure returns any query result sets, the result sets are returned to the caller of the stored procedure. If the SQL CALL statement is nested, the result sets are visible only to the program that is at the previous nesting level. For example, Figure 18 illustrates a scenario in which a client program calls stored procedure PROCA, which in turn calls stored procedure PROCB. Only PROCA can access any result sets that PROCB returns; the client program has no access to the query result sets. The number of query result sets that PROCB returns does not count toward the maximum number of query results that PROCA can return.

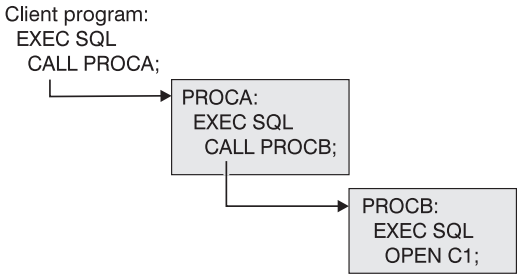


Figure 18. Nested CALL statements

Some stored procedures cannot be nested. A stored procedure, user-defined function, or trigger cannot call a stored procedure that is defined with the COMMIT ON RETURN attribute. Procedures that are defined with the AUTONOMOUS attribute cannot call other procedures that are defined with the AUTONOMOUS attribute.

Multiple calls to the same stored procedure: If the server and requester are both Version 8 or later of DB2 for z/OS (running in new-function mode), you can call a stored procedure multiple times within an application and at the same nesting level. Each call to the same stored procedure causes a unique instance of the stored procedure to run. If the stored procedure returns result sets, each instance of the stored procedure opens its own set of result set cursors.

The application might receive a "resource unavailable message" if the CALL statement causes the values of the maximum number of active stored procedures or maximum number open cursors to be exceeded. The value of field MAX STORED PROCEDURES (on installation panel DSNTIPX) defines the maximum number of active stored procedures that are allowed per thread. The value of field MAX OPEN CURSORS (on installation panel DSNTIPX) defines the maximum number of open cursors (both result set cursors and regular cursors) that are allowed per thread.

If you make multiple calls to the same stored procedure within an application, be aware of the following considerations:

- A DESCRIBE PROCEDURE statement describes the last instance of the stored procedure.
- The ASSOCIATE LOCATORS statement works on the last instance of the stored procedure.
- The ALLOCATE CURSOR statement must specify a unique cursor name for a result set returned from an instance of the stored procedure. Otherwise, you will lose the data from the result sets that are returned from prior instances or calls to the stored procedure.

You should issue an ASSOCIATE LOCATORS statement (or DESCRIBE PROCEDURE statement) after each call to the stored procedure to get a unique locator value for each result set.

Using host variables: If the CALL statement contains host variables, the contents of the host variables are assumed to be in the encoding scheme that was specified in the ENCODING parameter when the package or plan that contains the statement was bound.

Examples

Example 1: A PL/I application has been precompiled on DB2 ALPHA and a package was created at DB2 BETA with the BIND subcommand. A CREATE PROCEDURE statement was issued at BETA to define the procedure SUMARIZE, which allows nulls and has two parameters. The first parameter is defined as IN and the second parameter is defined as OUT. Some of the statements that the application that runs at DB2 ALPHA might use to call stored procedure SUMARIZE include:

```
EXEC SQL CONNECT TO BETA;
V1 = 528671;
IV = -1;
EXEC SQL CALL SUMARIZE(:V1,:V2 INDICATOR :IV);
```

Example 2: Suppose that stored procedure MYPROC exists and produces several result sets. An application might include statements like the following to access the result sets produced by MYPROC:

```
-- Invoke stored procedure MYPROC that returns several result sets
EXEC SQL CALL MYPROC (....);
-- Copy the locator values for the result sets into result set locator variables
EXEC SQL ASSOCIATE RESULT SET LOCATORS (:RS1, :RS2, :RS3) WITH PROCEDURE MYPROC;
-- Allocate cursors for the result set cursors
EXEC SQL ALLOCATE CSR1 CURSOR FOR RESULT SET :RS1;
EXEC SQL ALLOCATE CSR2 CURSOR FOR RESULT SET :RS2;
EXEC SQL ALLOCATE CSR3 CURSOR FOR RESULT SET :RS3;
-- Process data returned with the result set cursors
DO WHILE (SQLCODE = 0);
EXEC SQL FETCH CSR1 INTO .....
END;
EXEC SQL CLOSE CSR1;
-- do similar processing with other result sets
...
```

Example 3: Suppose that procedure FIND_CUSTOMERS has the following parameters:

- An IN parameter that is an array of phone numbers
- An IN parameter that is a prefix value to search for a match
- An OUT parameter that returns an array of phone numbers

FIND_CUSTOMERS searches the input array variable for phone numbers that match the prefix value, and returns an array that contains the phone numbers that match the prefix value.

FIND_CUSTOMERS looks like this:

```
-------------------------------------------------------------------------------
-- Create an SQL procedure with array parameters. The array parameters are defined
-- with the PHONENUMBERS array type.  The procedure searches for numbers in
-- IN_PHONENUMBERS that begin with the given prefix, and returns the phone numbers
-- in the NUMBERS_OUT parameter.
```

```
-------------------------------------------------------------------------------------
CREATE PROCEDURE FIND_CUSTOMERS(
  IN NUMBERS_IN PHONENUMBERS,
  IN PREFIX CHAR(3),
  OUT NUMBERS_OUT PHONENUMBERS)
BEGIN
  DECLARE I, J INTEGER;

  SET I = 1;
  SET J = 1;

-- Initialize NUMBERS_OUT to an empty array using an array constructor with no elements
  SET NUMBERS_OUT = ARRAY[ ];
  WHILE i < CARDINALITY(NUMBERS_IN) DO
    IF SUBSTR(NUMBERS_IN[I], 1, 3) = PREFIX THEN
      SET NUMBERS_OUT[J] = NUMBERS_IN[I];
      SET J = J + 1;
    END IF;
    SET I = I + 1;
  END WHILE;
END %
```

In the calling routine, declare array variables, and initialize the input array with values from an array constructor. Then invoke the procedure:

```
CREATE TYPE PHONENUMBERS AS VARCHAR(20) ARRAY[10]; -- Create an array type
DECLARE PNUMBER_ARRAY PHONENUMBERS;                -- Declare input array variable
DECLARE PNUMBER_ARRAY_OUT PHONENUMBERS;            -- Declare output array variable
SET  PNUMBER_ARRAY = ARRAY['416-305-3745',
                           '905-414-4565',
                           '416-305-3746'];
CALL FIND_CUSTOMERS(PNUMBER_ARRAY,      -- NUMBERS_IN parameter (IN parm)
                    '416',              -- PREFIX parameter (IN parm)
                    PNUMBER_ARRAY_OUT); -- NUMBERS_OUT parameter (OUT parm)
```

The CALL statement returns an array value with the following information in the argument corresponding to the NUMBERS_OUT parameter, which sets the PNUMBER_ARRAY_OUT variable:

```
['416-305-3745',
 '416-305-3746']
```

CLOSE

The CLOSE statement closes a cursor. If a temporary copy of a result table was created when the cursor was opened, that table is destroyed.

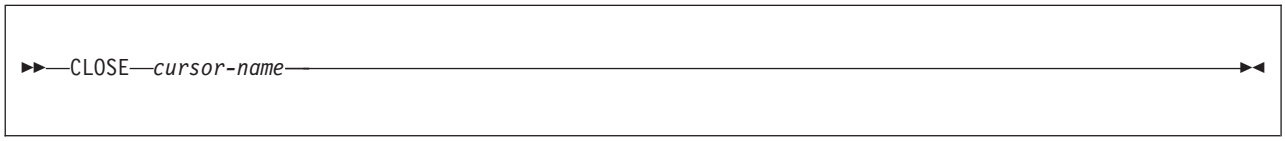
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

See "DECLARE CURSOR" on page 1535 for the authorization required to use a cursor.

Syntax



Description

cursor-name
Identifies the cursor to be closed. The cursor name must identify a declared cursor as explained in "DECLARE CURSOR" on page 1535. When the CLOSE statement is executed, the cursor must be in the open state.

Notes

Implicit cursor close: At the end of a unit of work, all open cursors declared without the WITH HOLD option that belong to an application process are implicitly closed.

Close cursors for performance: Explicitly closing cursors as soon as possible can improve performance.

Procedure considerations: Special rules apply to cursors within procedures that have not been closed before returning to the calling program. For more information, see "CALL" on page 1117.

Allocated cursors: The cursor could have been allocated. See "ALLOCATE CURSOR" on page 847.

Example

A cursor is used to fetch one row at a time into the application program variables DNUM, DNAME, and MNUM. Finally, the cursor is closed. If the cursor is reopened, it is again located at the beginning of the rows to be fetched.

```
EXEC SQL DECLARE C1 CURSOR FOR
   SELECT DEPTNO, DEPTNAME, MGRNO
   FROM DSN8B10.DEPT
   WHERE ADMRDEPT = 'A00'
   END-EXEC.
```

```
EXEC SQL OPEN C1 END-EXEC.

EXEC SQL FETCH C1 INTO :DNUM, :DNAME, :MNUM END-EXEC.

IF SQLCODE = 100
   PERFORM DATA-NOT-FOUND
ELSE
   PERFORM GET-REST-OF-DEPT
   UNTIL SQLCODE IS NOT EQUAL TO ZERO.

EXEC SQL CLOSE C1 END-EXEC.

GET-REST-OF-DEPT.
   EXEC SQL FETCH C1 INTO :DNUM, :DNAME, :MNUM END-EXEC.
```

COMMENT

The COMMENT statement adds or replaces comments in the descriptions of various objects in the DB2 catalog at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

For a comment on the following objects, the privilege set must include at least one of the listed authorities or privileges:

Table, view, index, column, or alias for a table or view:

- Ownership of the table, view, alias, or index
- DBADM authority for its database (tables and indexes only)
- SYSADM or SYSCTRL authority
- System DBADM
- SECADM authority (if the table has an activated row permission or column access control)

If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

Distinct type, stored procedure, trigger, or user-defined function:

- Ownership of the distinct type, stored procedure, trigger, or user-defined function
- The ALTERIN privilege on the schema (for the addition of comments)
- SYSADM or SYSCTRL authority
- System DBADM

Secure trigger or secure user-defined function:

- SECADM authority
- CREATE_SECURE_OBJECT privilege

Package:

- Ownership of the package
- The BINDAGENT privilege granted from the package owner
- PACKADM authority for the collection or for all collections
- SYSADM or SYSCTRL authority
- System DBADM

Role or a trusted context:

- Ownership of the object
- SYSADM or SYSCTRL authority
- SECADM

If the installation parameter SEPARATE SECURITY is NO, SYSADM authority has implicit SECADM and SYSCTRL authority and can drop a role or trusted context.

Sequence or alias for a sequence:

- Ownership of the sequence

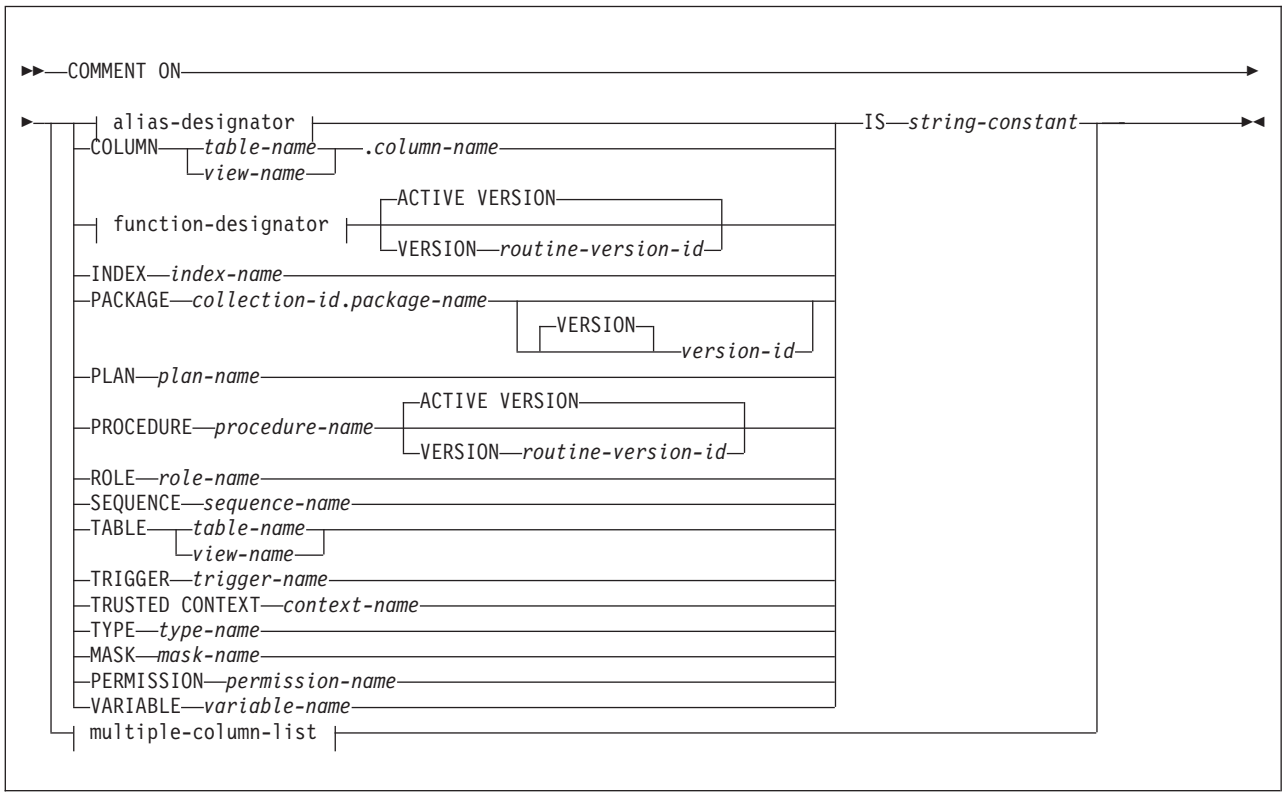
- The ALTER privilege for the sequence if the target is a sequence
- The ALTERIN privilege on the schema
- SYSADM or SYSCTRL authority
- System DBADM

The authorization ID that matches the schema name implicitly has the ALTERIN privilege on the schema.

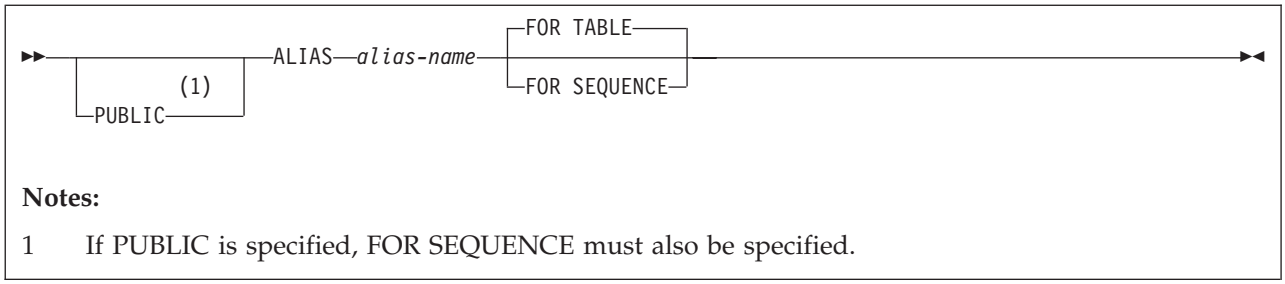
Row permission or column mask:
SECADM authority

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set is determined by the DYNAMICRULES behavior in effect (run, bind, define, or invoke) and is summarized in Table 94 on page 841. (For more information on these behaviors, including a list of the DYNAMICRULES bind option values that determine them, see "Authorization IDs and dynamic SQL" on page 75.)

Syntax



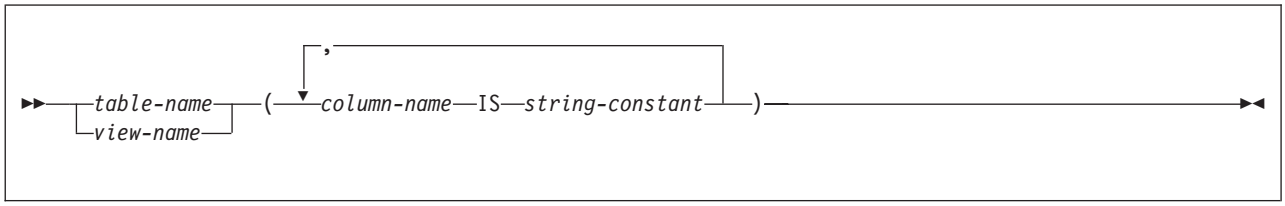
alias-designator



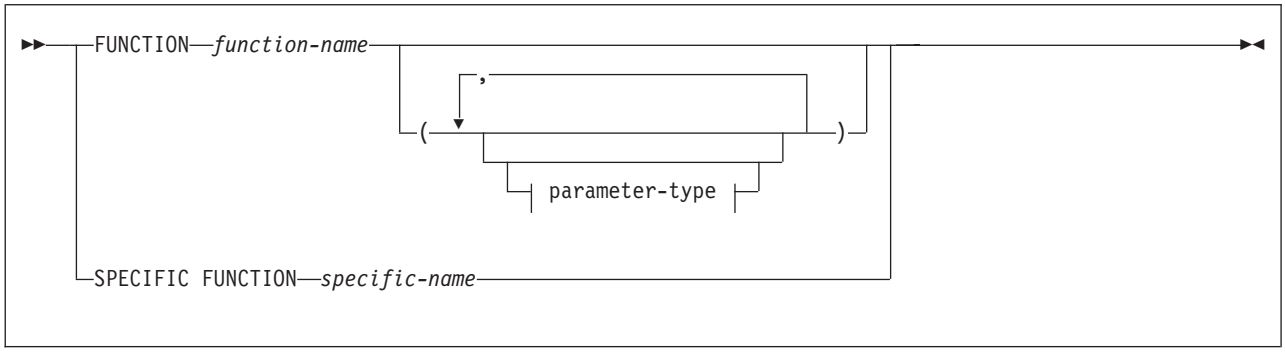
Notes:

1 If PUBLIC is specified, FOR SEQUENCE must also be specified.

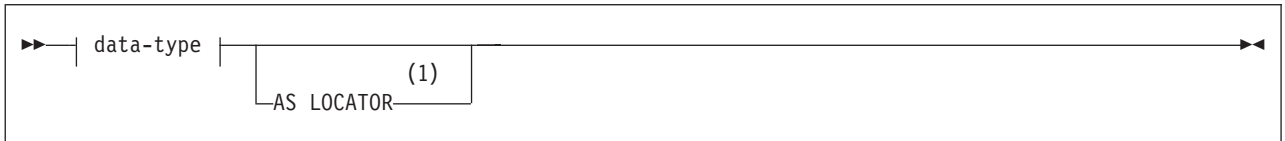
multiple-column-list



function-designator



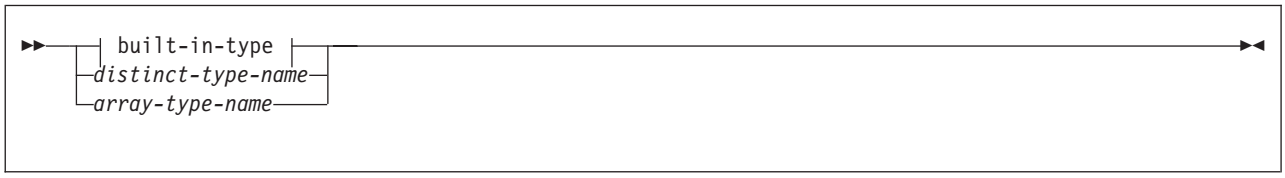
parameter-type



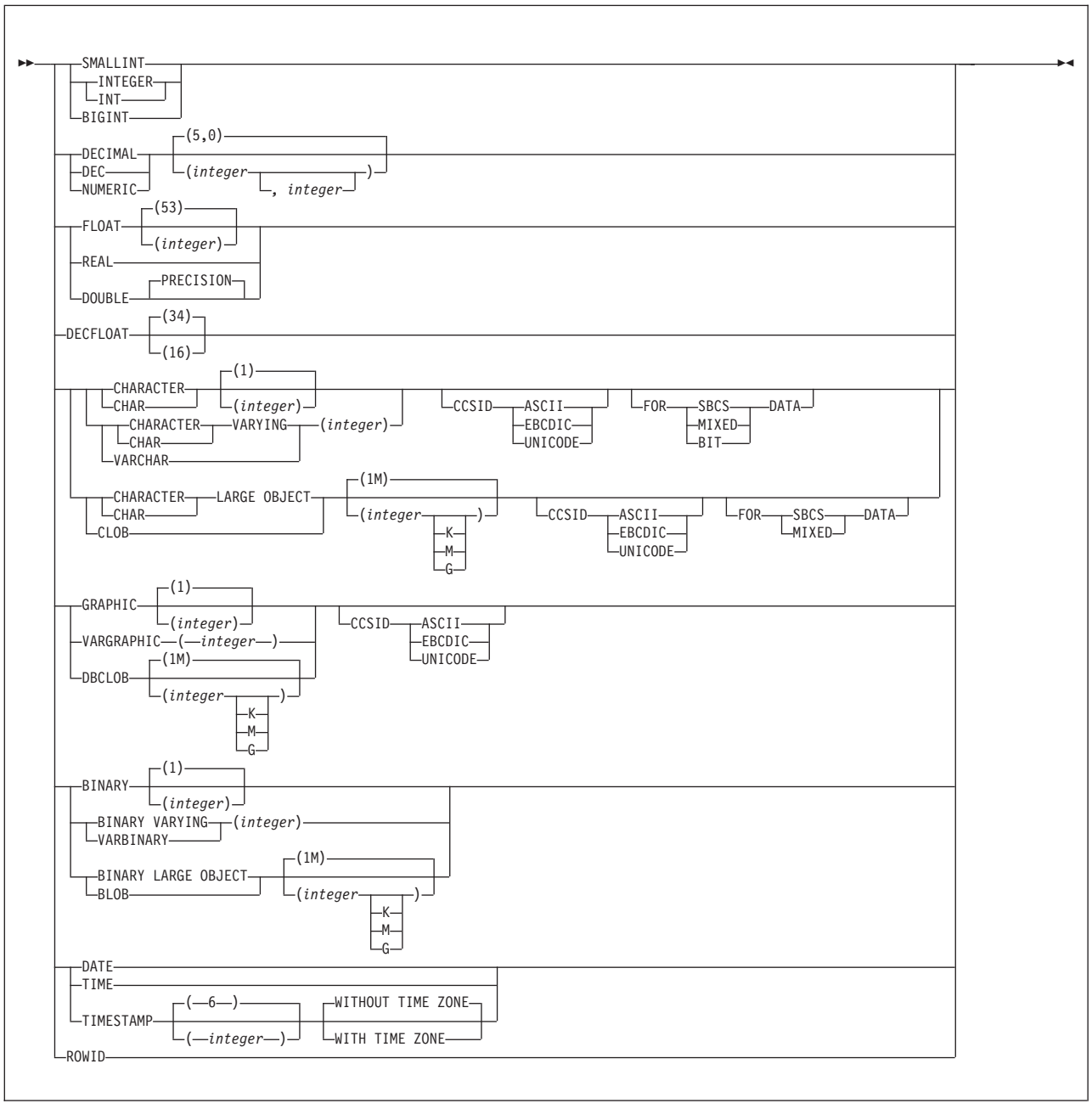
Notes:

1 AS LOCATOR can be specified only for a LOB data type or a distinct type that is based on a LOB data type.

data-type



built-in-type



Description

alias-designator

ALIAS *alias-name*
Indicates a comment will be added or replaced for an alias. The name must identify an alias that exists at the current server.

If the PUBLIC keyword is specified, *alias-name* must identify a public alias that exists at the current server. The comment is applied to a public alias.

FOR TABLE
Specifies that the alias is for a table or a view. The comment replaces the value of the REMARKS column of the SYSIBM.SYSTABLES catalog table for the row that describes the alias.

FOR SEQUENCE
Specifies that the alias is for a sequence. The comment replaces the value of the REMARKS column of the SYSIBM.SYSSEQUENCES catalog table for the row that describes the alias.

COLUMN *table-name.column-name* **or** *view-name.column-name*
Identifies the column to which the comment applies. The name must identify a column of a table or view that exists at the current server. The name must not identify a column of a declared temporary table. The comment is placed into the REMARKS column of the SYSIBM.SYSCOLUMNS catalog table, for the row that describes the column.

Do not use TABLE or COLUMN to comment on more than one column in a table or view. Give the table or view name and then, in parentheses, a list in the form:

```
column-name IS string-constant,
column-name IS string-constant,...
```

The column names must not be qualified, each name must identify a column of the specified table or view, and that table or view must exist at the current server.

FUNCTION or SPECIFIC FUNCTION
Identifies the function to which the comment applies. The function must exist at the current server, and it must be a function that was defined with the CREATE FUNCTION statement or a cast function that was generated by a CREATE TYPE statement. The comment is placed in the REMARKS column of the SYSIBM.SYSROUTINES catalog table for the row that describes the function.

The function can be identified by its name, function signature, or specific name. If the function was defined with a table parameter (the LIKE TABLE was specified in the CREATE FUNCTION statement to indicate that one of the input parameters is a transition table), you must identify the function with its function name, if it is unique, or with its specific name.

FUNCTION *function-name*
Identifies the function by its function name. There must be exactly one function with *function-name* in the schema. The function can have any number of input parameters. If the schema does not contain a function with *function-name*, or if the schema contains more than one function with this name, and error is returned.

FUNCTION *function-name* **(***parameter-type***,...)**
Identifies the SQL function by its function signature, which uniquely identifies the function. A function with the function signature must exist in the explicitly or implicitly specified schema.

If *function-name*() is specified, the function that is identified must have zero parameters.

function-name
Identifies the name of the function. If the function was defined with a table parameter (the **LIKE TABLE** *name* **AS LOCATOR** clause was specified in the CREATE FUNCTION statement to indicate that one of the input parameters is a transition table), the function signature

cannot be used to uniquely identify the function. Instead, use one of the other syntax variations to identify the function with its function name, if unique, or with its specific name.

(*parameter-type***,...)**
Specifies the number of input parameters of the function and the name and data type of each parameter.

(*data-type***,...)**
Identifies the number of input parameters of the function and the data type of each parameter. The data type of each parameter must match the data type that was specified in the CREATE FUNCTION statement for the parameter in the corresponding position. The number of data types and the logical concatenation of the data types are used to uniquely identify the function.

For data types that have a length, precision, or scale attribute, you can use a set of empty parentheses, specify a value, or accept the default values:

- Empty parentheses indicate that DB2 is to ignore the attribute when determining whether the data types match.

  For example, DEC() will be considered a match for a parameter of a function defined with a data type of DEC(7,2). Similarly DECFLOAT() will be considered a match for DECFLOAT(16) or DECFLOAT(34).

  FLOAT cannot be specified with empty parentheses because its parameter value indicates different data types (REAL or DOUBLE).
- If you use a specific value for a length, precision, or scale attribute, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

  The specific value for FLOAT(*n*) does not have to exactly match the defined value of the source function because `1<=`*n*`<= 21` indicates REAL and `22<=`*n*`<=53` indicates DOUBLE. Matching is based on whether the data type is REAL or DOUBLE.
- If length, precision, or scale is not explicitly specified and empty parentheses are not specified, the default length of the data type is implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

For data types with a subtype or encoding scheme attribute, specifying the **FOR** *subtype* **DATA** clause or the **CCSID** clause is optional. Omission of either clause indicates that DB2 is to ignore the attribute when determining whether the data types match. If you specify either clause, it must match the value that was implicitly or explicitly specified in the CREATE FUNCTION statement.

AS LOCATOR
Specifies that the function is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or a distinct type based on a LOB.

SPECIFIC FUNCTION *specific-name*
Identifies a particular user-defined function by its specific name. The name is implicitly or explicitly qualified with a schema name. A function with the specific name must exist in the schema. If the specific name is not

qualified, it is implicitly qualified with a schema name as described in the description for **FUNCTION** *function-name*.

ACTIVE VERSION
Specifies that the comment applies to the currently active version of the routine that is specified by *function-name*.

ACTIVE VERSION is the default.

VERSION *routine-version-id*
Specifies that the comment applies only to the version of the routine that is identified by *routine-version-id*. *routine-version-id* must identify a version of the specified routine that already exists at the current server. If *routine-version-id* is not specified, a null string is used as the version identifier.

INDEX *index-name*
Identifies the index to which the comment applies. *index-name* must identify an index that exists at the current server. The comment is placed in the REMARKS column of the SYSIBM.SYSINDEXES catalog table for the row that describes the index.

MASK *mask-name*
Identifies the column mask to which the comment applies. *mask-name* must identify a column mask that exists at the current server. The comment is placed in the REMARKS column of the SYSIBM.SYSCONTROLS catalog table for the row that describes the column mask.

PACKAGE *collection-id.package-name*
Identifies the package to which the comment applies. You must qualify the package name with a collection ID. *collection-id.package-name* must identify a package that exists at the current server. The name plus the implicitly or explicitly specified *version-id* must identify a package that exists at the current server. Omission of the *version-id* is an implicit specification of the null version.

The name must not identify a trigger package or a package that is associated with an SQL routine. Specify this clause to comment on a package that was created as the result of a BIND COPY command used to deploy a version of a native SQL procedure.

VERSION *version-id*
version-id is the version identifier that was assigned to the package's DBRM when the DBRM was created. If *version-id* is not specified, a null version is used as the version identifier.

Delimit the version identifier when it:

- Is generated by the VERSION(AUTO) precompiler option
- Begins with a digit
- Contains lowercase or mixed-case letters

For more on version identifiers, see the information on preparing an application program for execution in *DB2 Application Programming and SQL Guide*.

PERMISSION *permission-name*
Identifies the row permission to which the comment applies. *permission-name* must identify a row permission that exists at the current server. The comment is placed in the REMARKS column of the SYSIBM.SYSCONTROLS catalog table for the row that describes the row permission.

PLAN *plan-name*
Identifies the plan to which the comment applies. *plan-name* must identify a plan that exists at the current server.

PROCEDURE *procedure-name*
Identifies the procedure to which the comment applies. *procedure-name* must identify a procedure that exists at the current server.

ACTIVE VERSION
Specifies that the comment applies to the currently active version of the routine that is specified by *procedure-name*.

ACTIVE VERSION is the default.

VERSION *routine-version-id*
Specifies that the comment applies only to the version of the routine that is identified by *routine-version-id*. *routine-version-id* must identify a version of the specified routine that already exists at the current server. If *routine-version-id* is not specified, a null string is used as the version identifier.

ROLE *role-name*
Identifies the role to which the comment applies. *role-name* must identify a role that exists at the current server. The comment is placed in the REMARKS column of the SYSIBM.SYSROLES catalog table for the row that describes the role.

SEQUENCE *sequence-name*
Identifies the sequence to which the comment applies.

sequence-name must identify a sequence that exists at the current server. *sequence-name* must not be the name of an internal sequence object that is used by DB2. The comment is placed in the REMARKS column of the SYSIBM.SYSSEQUENCES catalog table for the row that describes the sequence.

TABLE *table-name* **or** *view-name*
Identifies the table or view to which the comment applies. *table-name* or *view-name* must identify a table, auxiliary table, or view that exists at the current server. *table-name* must not identify a declared temporary table. The comment is placed in the REMARKS column of the SYSIBM.SYSTABLES catalog table for the row that describes the table or view.

TRIGGER *trigger-name*
Identifies the trigger to which the comment applies. *trigger-name* must identify a trigger that exists at the current server. The comment is placed in the REMARKS column of the SYSIBM.SYSTRIGGERS catalog table for the row that describes the trigger.

TRUSTED CONTEXT *context-name*
Identifies the trusted context to which the comment applies. *context-name* must identify a trusted context that exists at the current server. The comment is placed in the REMARKS column of the SYSIBM.SYSCONTEXT catalog table for the row that describes the trusted context.

TYPE *type-name*
Identifies the user-defined type to which the comment applies. *type-name* must identify a user-defined type that exists at the current server. The comment is placed in the REMARKS column of the SYSIBM.SYSDATATYPES catalog table for the row that describes the user-defined type.

VARIABLE *variable-name*
Identifies the global variable to which the comment applies. *variable-name* must identify a global variable that exists at the current server. *variable-name* must not identify a built-in global variable.

IS *string-constant*
Introduces the comment that you want to make. *string-constant* can be any SQL character string constant of up to 762 bytes.

multiple-column-list
To comment on more than one column in a table or view with a single COMMENT statement, specify the table or view name, followed by a list in parentheses of the form:

```
(column-name IS string-constant,
 column-name IS string-constant,
  ...)
```

Each column name must not be qualified, and must identify a column of the specified table or view that exists at the current server.

Notes

Alternative syntax and synonyms:
To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following syntax alternatives:

- DATA TYPE or DISTINCT TYPE as a synonym for TYPE
- COMMENT ON ALIAS SYSPUBLIC.*name* can be specified as an alternative to COMMENT ON PUBLIC ALIAS SYSPUBLIC.*name*

Examples

Example 1: Enter a comment on table DSN8B10.EMP.

```
COMMENT ON TABLE DSN8B10.EMP
  IS 'REFLECTS 1ST QTR 81 REORG';
```

Example 2: Enter a comment on view DSN8B10.VDEPT.

```
COMMENT ON TABLE DSN8B10.VDEPT
  IS 'VIEW OF TABLE DSN8B10.DEPT';
```

Example 3: Enter a comment on the DEPTNO column of table DSN8B10.DEPT.

```
COMMENT ON COLUMN DSN8B10.DEPT.DEPTNO
  IS 'DEPARTMENT ID - UNIQUE';
```

Example 4: Enter comments on the two columns in table DSN8B10.DEPT.

```
COMMENT ON DSN8B10.DEPT
  (MGRNO IS 'EMPLOYEE NUMBER OF DEPARTMENT MANAGER',
   ADMRDEPT IS 'DEPARTMENT NUMBER OF ADMINISTERING DEPARTMENT');
```

Example 5: Assume that you are SMITH and that you created the distinct type DOCUMENT in schema SMITH. Enter comments on DOCUMENT.

```
COMMENT ON TYPE DOCUMENT
  IS 'CONTAINS DATE, TABLE OF CONTENTS, BODY, INDEX, and GLOSSARY';
```

Example 6: Assume that you are SMITH and you know that ATOMIC_WEIGHT is the only function with that name in schema CHEM. Enter comments on ATOMIC_WEIGHT.

```
COMMENT ON FUNCTION CHEM.ATOMIC_WEIGHT
  IS 'TAKES ATOMIC NUMBER AND GIVES ATOMIC WEIGHT';
```

Example 7: Assume that you are SMITH and that you created the function CENTER in schema SMITH. Enter comments on CENTER, using the signature to uniquely identify the function instance.

```
COMMENT ON FUNCTION CENTER (INTEGER, FLOAT)
  IS 'USES THE CHEBYCHEV METHOD';
```

Example 8: Assume that you are SMITH and that you created another function named CENTER in schema JOHNSON. You gave the function the specific name FOCUS97. Enter comments on CENTER, using the specific name to identify the function instance.

```
COMMENT ON SPECIFIC FUNCTION JOHNSON.FOCUS97
  IS 'USES THE SQUARING TECHNIQUE';
```

Example 9: Assume that you are SMITH and that procedure OSMOSIS is in schema BIOLOGY. Enter comments on OSMOSIS. Your comments will apply to the currently active version of the procedure OSMOSIS.

```
COMMENT ON PROCEDURE BIOLOGY.OSMOSIS
  IS 'CALCULATIONS THAT MODEL OSMOSIS';
```

Example 11: Assume that you are SMITH and that trigger BONUS is in your schema. Enter comments on BONUS.

```
COMMENT ON TRIGGER BONUS
  IS 'LIMITS BONUSES TO 10% OF SALARY';
```

Example 12: Provide a comment for package MYPKG, which is in collection COLLIDA.

```
COMMENT ON COLLIDA.MYPKG
  IS 'THIS IS MY PACKAGE';
```

Example 14: Provide a comment on role ROLE1:

```
COMMENT ON ROLE ROLE1
  IS 'Role defined for trusted context, ctx1';
```

Example 15: Provide a comment on trusted context CTX1:

```
COMMENT ON TRUSTED CONTEXT CTX1
  IS 'WEBSPHERE SERVER';
```

Example 15: Provide a comment on column mask M1:

```
COMMENT ON MASK M1
  IS 'Column mask for column EMP.SALARY';
```

COMMIT

The COMMIT statement ends the unit of recovery in which it is executed and a new unit of recovery is started for the process. The statement commits all changes made by SQL schema statements and SQL data change statements during the unit of work.

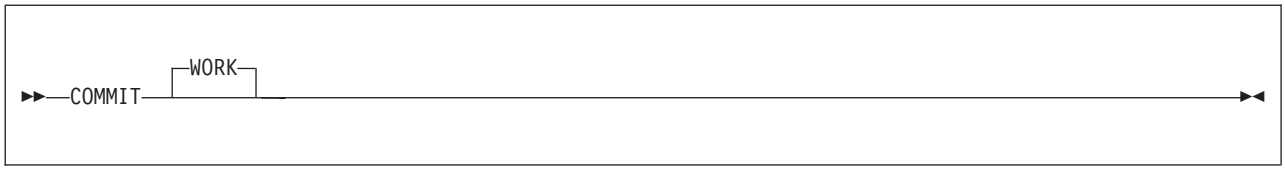
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared. It cannot be used in the IMS or CICS environment.

Authorization

None required.

Syntax



Description

The COMMIT statement ends the unit of recovery in which it is executed and a new unit of recovery is started for the process. The statement commits all changes made by SQL schema statements and SQL data change statements during the unit of work. For more information see Chapter 5, “Statements,” on page 833.

Notes

Recommended coding practices: Code an explicit COMMIT or ROLLBACK statement at the end of an application process. Either an implicit commit or rollback operation will be performed at the end of an application process depending on the application environment. Thus, a portable application should explicitly execute a COMMIT or ROLLBACK statement before execution ends in those environments where explicit COMMIT or ROLLBACK is permitted.

Effect of COMMIT: All savepoints that are set within the unit of recovery are released, and all changes are committed for the following statements that are executed during the unit of recovery:

- ALTER
- COMMENT
- CREATE
- DELETE
- DROP
- EXPLAIN
- GRANT
- INSERT
- LABEL

- MERGE
- RENAME
- REVOKE
- UPDATE
- SELECT INTO with an SQL data change statement
- subselect with an SQL data change statement

SQL connections are ended when any of the following conditions apply:

- The connection is in the release pending status
- The connection is not in the release pending status but it is a remote connection and:
  - The DISCONNECT(AUTOMATIC) bind option is in effect, or
  - The DISCONNECT(CONDITIONAL) bind option is in effect and an open WITH HOLD cursor is not associated with the connection.

For existing connections, all LOB locators are disassociated, except for those locators for which a HOLD LOCATOR statement has been issued without a corresponding FREE LOCATOR statement. All open cursors that were declared without the WITH HOLD option are closed. All open cursors that were declared with the WITH HOLD option are preserved, along with any SELECT statements that were prepared for those cursors.

Static and dynamic INSERT, UPDATE, DELETE, and MERGE statements that reference declared global temporary tables that were defined without ON COMMIT DROP TABLE and are bound with or use the RELEASE(DEALLOCATE) option are kept past commit points. The statement is not kept across the commit point if one of the following conditions is true:

- The declared global temporary table is defined with the ON COMMIT DROP TABLE option.
- The statement uses the RELEASE(COMMIT) bind option.
- The statement also references a DB2 base object (for example, a table or view), and one of the following statements is true:
  - The base object reference is for a DB2 catalog table.
  - At the commit point, DB2 determines that another DB2 thread is waiting for an X-lock on the base object's database descriptor (DBD).
  - The statement references an XML function or operation, and at the commit point DB2 determines that the base object DBD S-lock for the XML operation must be released.
  - At the commit point, DB2 determines that a base object DBD S-lock that is used by the statement must be released and cannot be maintained across the commit point.
- DB2 determines that another DB2 thread is waiting for an X-lock on the DB2 package that contains the statement.

Prepared dynamic statements are kept past commit points if one of the following conditions is true:

- Dynamic caching is enabled for your system. In that case, all prepared SELECT and data change statements that are bound with KEEPDYNAMIC(YES) are kept past the commit point.
- The statements reference a declared global temporary table that was defined without ON COMMIT DROP TABLE, and the package was bound with or uses

the RELEASE(DEALLOCATE) option. In that case, all prepared INSERT, UPDATE, DELETE, and MERGE statements that reference the declared global temporary table are kept across the commit point.

Prepared statements cannot be kept past a commit if any of the following conditions is true:

- SQL RELEASE has been issued for that site.
- Bind option DISCONNECT(AUTOMATIC) was used.
- Bind option DISCONNECT(CONDITIONAL) was used and there are no open WITH HOLD cursors for that site.
- The statement references a declared global temporary table, has no open WITH HOLD cursor, and is in a package that is bound with the RELEASE(COMMIT) option.
- The statement references a declared global temporary table that was defined with the ON COMMIT DROP TABLE option. The statement also has no open WITH HOLD cursor, and the statement's package is bound with or uses the RELEASE(DEALLOCATE) option.

All implicitly acquired locks are released, except for the following locks:

- Locks that are required for the cursors that were not closed
- Table and table space locks when the RELEASE parameter on the bind command was not RELEASE(COMMIT)
- LOB locks and LOB table space locks that are required for held LOB locators

For an explanation of the duration of explicitly acquired locks, see *DB2 Performance Monitoring and Tuning Guide*.

All rows of every created temporary table of the application process are deleted with the exception that the rows of a created temporary table are not deleted if any program in the application process has an open WITH HOLD cursor that is dependent on that table. In addition, if RELEASE(COMMIT) is in effect, the logical work files for the created temporary tables whose rows are deleted are also deleted.

All rows of every declared temporary table of the application process are deleted with these exceptions:

- The rows of a declared temporary table that is defined with the ON COMMIT PRESERVE ROWS attribute are not deleted.
- The rows of a declared temporary table that is defined with the ON COMMIT DELETE ROWS attribute are not deleted if any program in the application process has an open WITH HOLD cursor that is dependent on that table.

Implicit commit operations: In all DB2 environments, the normal termination of a process is an implicit commit operation.

Restrictions on the use of COMMIT: The COMMIT statement cannot be used in the IMS or CICS environment. To cause a commit operation in these environments, SQL programs must use the call prescribed by their transaction manager. The effect of these commit operations on DB2 data is the same as that of the SQL COMMIT statement.

The COMMIT statement cannot be used in a stored procedure if the procedure is in the calling chain of a user-defined function or a trigger or DB2 is not the commit coordinator.

Effect of commit on special registers: Issuing a COMMIT statement may cause special registers to be re-initialized. Whether one of these special registers is affected by a commit depends on whether the special register has been explicitly set within the application process. For example, assume that the PATH special register has not been explicitly set with a SET PATH statement in the application process. After a commit, the value of PATH is re-initialized. For information on the initialization of PATH, which can take the current value of CURRENT SQLID into consideration, see "CURRENT PATH" on page 184.

Effect of commit on global variables: Global variables are not controlled at the transaction level. Issuing a COMMIT statement does not effect the contents of a global variable.

Example

Commit all DB2 database changes made since the unit of recovery was started.

```
COMMIT WORK;
```

CONNECT

The CONNECT statement connects an application process to a database server. This server becomes the *current server* for the process. The CONNECT statement of DB2 for z/OS is equivalent to CONNECT (Type 2) in IBM DB2 SQL Reference for Cross-Platform Development.

Refer to "Distributed data" on page 35 for complete information about connections, the current server, commit processing, and distributed and remote units of work.

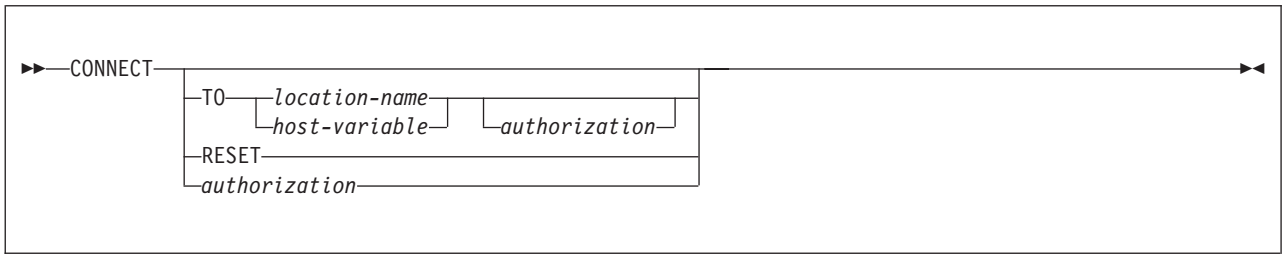
Invocation

This statement can only be embedded within an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

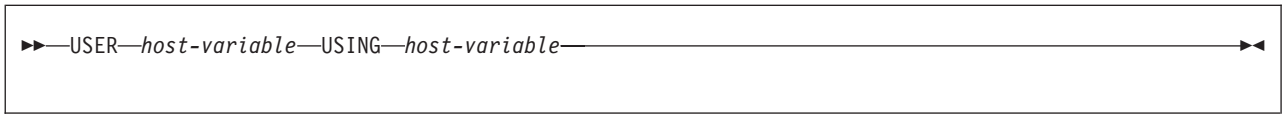
Authorization

The primary authorization ID of the process or the authorization ID that is specified in this statement must be authorized to connect to the specified server. The server performs the authorization check when the statement is executed, and determines the specific authorization that is required. See *DB2 Administration Guide* for further information.

Syntax



authorization:



Description

TO *location-name* **or** *host-variable*
Identifies the server by the specified location name or by the location name that is contained in the host variable. If a host variable is specified:

- It must be a CHAR or VARCHAR variable with a length attribute that is not greater than 16. (A C NUL-terminated character string can be up to 17 bytes long.)
- It must not be followed by an indicator variable.
- The location name must be left-justified within the host variable and must conform to the rules for forming an ordinary identifier.
- If the length of the location name is less than the length of the host variable, it must be padded on the right with blanks.

- It must not contain lowercase characters.
- If used with an SQL procedure language application, host variable must be a qualified SQL-variable name or a qualified SQL-parameter name.

When the CONNECT statement is executed:

- The location name must identify a server known to the local DB2 subsystem. Hence, the location name must be the location name of the local DB2 subsystem or it must appear in the LOCATION column of the SYSIBM.LOCATIONS table.
- The application process must not have an existing connection to the specified server, if the SQLRULES(STD) bind option is in effect.
- The application process must be in a connectable state, if the transaction is participating in a remote unit of work.

RESET
CONNECT RESET is equivalent to CONNECT TO *x* where *x* is the location name of the local DB2 subsystem.

- If the SQLRULES(DB2) bind option is in effect, CONNECT RESET establishes the local DB2 subsystem as the current SQL connection.
- If the SQLRULES(STD) bind option is in effect, CONNECT RESET establishes the local DB2 subsystem as the current SQL connection only if the connection does not exist.

authorization
Specifies an authorization ID and a password that is used to verify that the authorization ID is authorized to connect to the server. Authorization cannot be specified when the connection type is IMS or CICS for a connection to the local DB2 subsystem. An attempt to do so causes an SQL error.

USER *host-variable*
Identifies the authorization name to use when connecting to the server. The value of *host-variable* must satisfy the following rules:

- The value must be a CHAR or VARCHAR variable with a length attribute that is not greater than 128.
- The value must be left-justified within the host variable and must conform to the rules for forming an authorization name.
- The value must not be followed by an indicator variable.
- The value must be padded on the right with blanks if the length of the authorization name is less than the length of the host variable.

For a connection to the local DB2 subsystem, a user ID that is longer than 8 characters causes an SQL error.

USING *host-variable*
Identifies the password of the authorization name to use when connecting to the server. The value of *host-variable* must satisfy the following rules:

- The value must be a CHAR or VARCHAR variable with a length attribute that is not greater than 128.
- The value must be left-justified.
- The value must not include an indicator variable.
- The value must be padded on the right with blanks if the length of the password is less than the length of the host variable.
- The value must not contain lowercase characters.

For a connection to a DB2 subsystem, a password that is longer than 8 characters causes an SQL error.

CONNECT USER/USING is equivalent to CONNECT TO *x* USER/USING where *x* is the location name of the local DB2 subsystem (which has the semantic of CONNECT RESET).

CONNECT with no operand
This form of the CONNECT statement returns information about the current server in the SQLERRP field of the SQLCA. SQLERRP returns blanks if the application process is in the unconnected state.

Executing a CONNECT with no operand has no effect on connection states.

In a remote unit of work, this form of CONNECT does not require the application process to be in a connectable state.

Notes

Successful connection: With the exception of a CONNECT with no operand statement, if execution of the CONNECT statement is successful:

- One of the following scenarios takes place in a **distributed unit of work**:
  - If the location name does not identify a server to which the application process is already connected, an SQL connection to the server is created and placed in the current and held state. The previously current SQL connection, if any, is placed in the dormant state.
  - If the location name identifies a server to which the application process is already connected, the associated SQL connection is dormant, and the SQLRULES(DB2) option is in effect, the SQL connection is placed in the current state. The previously current SQL connection, if any, is placed in the dormant state.
  - If the location name identifies a server to which the application process is already connected, the associated SQL connection is current, and the SQLRULES(DB2) option is in effect, the states of all SQL connections of the application process are unchanged.
- The following actions occur in a **remote unit of work**:
  - The application process is connected to the specified server.
  - An existing SQL connection of the application process is ended. As a result, all cursors of that SQL connection are closed, all prepared statements of that connection are destroyed, and so on.
- The location name is placed in the CURRENT SERVER special register.
- When CONNECT is used to connect back to the local DB2 subsystem, the CURRENT SQLID special register is reinitialized if the USER/USING clause is specified.
-
- Information about the server is placed in the SQLERRP field of the SQLCA. If the server is a DB2 product, the information has the form *pppvvrrm*, where:
  - *ppp* is:
    ARI for DB2 Server for VSE & VM
    DSN for DB2 for z/OS
    QSQ for DB2 for i
    SQL for DB2 for Linux, UNIX, and Windows
  - *vv* is a two-digit version identifier such as '11'.
  - *rr* is a two-digit release identifier such as '01'.
  - *m* is a one-digit modification level.

- Values 0, 1, 2, 3, and 4 are reserved for modification levels in conversion and enabling-new-function mode from Version 10 (CM10, CM10*, ENFM10, and ENFM10*)
- Values 5, 6, 7, 8, and 9 are for modification levels in new-function mode.

For example, if the server is Version 9 of DB2 for z/OS in new-function mode with the first level of maintenance, the value of SQLERRP is 'DSN09015'.

- Additional information about the connection is placed in the SQLERRMC field of the SQLCA. The contents are product-specific.

  Tip: Use the GET DIAGNOSTICS statement to get detailed diagnostic information about the last SQL statement that was executed.

Unsuccessful connection: With the exception of a CONNECT with no operand statement, if execution of the CONNECT statement is unsuccessful:

- In a **distributed unit of work**, the connection state of the application process and the states of its SQL connections are unchanged unless the failure was because an authorization check failed. If this is the case, the connection is placed in the connectable and unconnected state.
- In a **remote unit of work**, the SQLERRP field of the SQLCA is set to the name of the DB2 requester module that detected the error.

  If execution of the CONNECT statement is unsuccessful because the application process is not in the connectable state, the connection state of the application process is unchanged. If execution of the CONNECT statement is unsuccessful for any other reason, CURRENT SERVER is set to blanks and the application process is placed in the connectable and unconnected state.

Authorization: If the server is a DB2 subsystem, a user is authenticated in the following way:

- DB2 invokes RACF via the RACROUTE macro with REQUEST=VERIFY to verify the password.
- If the password is verified, DB2 then invokes RACF again via the RACROUTE macro with REQUEST=AUTH, to check whether the authorization ID is allowed to use DB2 resources defined to RACF.
- DB2 then invokes the connection exit routine if one has been defined.
- The connection then has a primary authorization ID, possibly one or more secondary IDs, and an SQL ID.

If the server is a remote DB2 subsystem, the requester generates authentication tokens and sends them to the remote site in the following way:

- The SECURITY_OUT column in SYSIBM.LUNAMES for SNA or the SECURITY_OUT column in SYSIBM.IPNAMES for TCP/IP must have one of the following values:
  - 'A' (already verified)
  - 'D' (userid and security-sensitive data encryption; TCP/IP only)
  - 'E' (userid, password, and security-sensitive data encryption; TCP/IP only)
  - 'P' (password)

  When the value is 'A', the user ID and password specified on the CONNECT is still sent.

  When the value is 'D', 'E', 'or 'P', the requester encrypts the user ID and password specified on the CONNECT for TCP/IP. However, if the Integrated Cryptographic Service Facility (ICSF) is not configured at the requester or if the server does not support encryption, one of the following actions occurs:

  - If the value of SECURITY_OUT in SYSIBM.IPNAMES is 'D' or 'E', SQLCODE -904 is returned if ICSF is not configured at the requester, and SQLCODE -30082 is returned if the server does not support encryption.
  - If the value of SECURITY_OUT in SYSIBM.IPNAMES is 'P', the requester does not encrypt the user ID and password and flows the tokens in clear text.
- For SNA, the ENCRYPTPSWDS column in SYSIBM.SYSLUNAMES must be not contain 'Y'.
- The authorization ID and password are verified at the server.
- In all cases, outbound translation—as specified in SYSIBM.USERNAMES—is not done.

Distributed unit of work: In general, the following are true:

- A CONNECT statement with the TO clause and the USER/USING clause can be executed only if no current or dormant connection to the named server exists. However, if the named server is the local DB2 subsystem and the CONNECT statement is the first SQL statement that is executed after the DB2 thread is created, the CONNECT statement executes successfully.
- A CONNECT statement without the TO clause but with the USER/USING clause can be executed only if no current or dormant connection to the local DB2 subsystem exists. However, if the CONNECT statement is the first SQL statement that is executed after the DB2 thread is created, the CONNECT statement executes successfully.

Remote unit of work: If the authorization check fails, the connection is placed in the connectable and unconnected state.

Precompiler options: Regardless of whether a program is precompiled with the CONNECT(1) or CONNECT(2) option, DB2 for z/OS negotiates with the remote server during the connection process to determine how to perform commits. If the remote server does not support the two-phase commit protocol, DB2 downgrades to perform one-phase commits.

Programs containing CONNECT statements that are precompiled with different CONNECT precompiler options cannot execute as part of the same application process. An error occurs when an attempt is made to execute the invalid CONNECT statement.

Host variables: If a CONNECT statement contains host variables, the contents of the host variables are assumed to be in the encoding scheme that was specified in the ENCODING parameter when the package or plan that contains the statement was bound.

Error processing: A CONNECT statement can return and indicate a successful execution even when no physical connection yet exists. DB2 delays the physical connection process, when possible, to economize on the number of messages it sends to a server. Therefore, errors in CONNECT statement processing can be reported following the next executable SQL statement, not immediately following the CONNECT statement.

Restrictions on array types and array variables: In any SQL statement other than a CALL statement, array types and array variables must not be referenced after a connection at a remote server has been established. This restriction includes an SQL statement that executes at a remote server as a result of a three-part name or alias that resolves to an object at a remote server. An exception is that an array

element can be the target of a FETCH, SELECT INTO, SET *assignment-statement*, or VALUES INTO statement in an SQL routine even when the statement is executed at a remote server.

Examples

Example 1: Connect an application to a DBMS. The location name is in the character-string variable *LOCNAME*, the authorization identifier is in the character-string variable *AUTHID*, and the password is in the character-string variable *PASSWORD*.

```
EXEC SQL CONNECT TO :LOCNAME USER :AUTHID USING :PASSWORD;
```

Example 2: Obtain information about the current server.

```
EXEC SQL CONNECT;
```

Example 3: Execute SQL statements in a distributed unit of work. The first CONNECT statement creates a connection to the EASTDB server. The second CONNECT statement creates a connection to the WESTDB server, and places the SQL connection to EASTDB in the dormant state.

```
EXEC SQL CONNECT TO EASTDB;
  -- execute statements referencing objects at EASTDB
EXEC SQL CONNECT TO WESTDB;
  -- execute statements referencing objects at WESTDB
```

Example 4: Connect the application to a DBMS whose location identifier is in the character-string variable LOC using the authorization identifier in the character-string variable AUTHID and the password in the character-string variable PASSWORD. Perform work for the user, and then release the connection and connect again using a different user ID and password.

```
EXEC SQL CONNECT TO :LOC USER :AUTHID USING :PASSWORD;
  -- execute SQL statements accessing data on the server
RELEASE :LOC;
EXEC SQL COMMIT;
  -- set AUTHID and PASSWORD to new values
EXEC SQL CONNECT TO :LOC USER :AUTHID USING :PASSWORD;
  -- execute SQL statements accessing data on the server
```

Example 5: Change servers in a remote unit of work. Assume that the application connected to a remote DB2 server, opened a cursor, and fetched rows from the cursor's result table. Subsequently, to connect to the local DB2 subsystem, the application executes the following statements:

```
EXEC SQL COMMIT WORK;
EXEC SQL CONNECT RESET;
```

The COMMIT is required because opening the cursor caused the application to enter the unconnectable and connected state.

If the cursor was declared with the WITH HOLD clause and was not closed with a CLOSE statement, it would still be open even after execution of the COMMIT statement. However, it would be closed with the execution of the CONNECT statement.

Related concepts:

Explicit CONNECT statements (Introduction to DB2 for z/OS)

"Distributed data" on page 35

Related tasks:

Accessing distributed data by using explicit CONNECT statements (DB2 Application programming and SQL)

Reusing a local trusted connection through the SQL CONNECT statement (Managing Security)

CREATE ALIAS

The CREATE ALIAS statement defines an alias for a table, a view, or a sequence. The definition is recorded in the DB2 catalog at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

To create an alias, the privilege set must include at least one of the listed authorities or privileges:

To create an alias for a table or a view:

- The CREATEALIAS privilege
- SYSADM or SYSCTRL authority
- DBADM or DBCTRL authority on the database that contains the table, if the alias is for a table and the value of field DBADM CREATE AUTH on installation panel DSNTIPP is YES
- System DBADM

To create an alias for a sequence:

- The CREATEIN privilege on the schema
- SYSADM or SYSCTRL authority
- System DBADM

If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the owner of the plan or package is a role, this role must hold the privileges for the privilege set. If the specified alias name includes a qualifier that is not the same as this authorization ID, the privilege set must include one of the following authorities:

- SYSADM or SYSCTRL authority
- DBADM or DBCTRL authority on the database that contains the table, if the alias is for a table and the value of field DBADM CREATE AUTH on installation panel DSNTIPP is YES

If ROLE AS OBJECT OWNER is in effect, the schema qualifier must be the same as the role, unless the role has the CREATEIN privilege on the schema, SYSADM authority, or SYSCTRL authority.

If ROLE AS OBJECT OWNER is not in effect, one of the following rules applies:

- If the privilege set lacks the CREATIN privilege on the schema, SYSADM authority, or SYSCTRL authority, the schema qualifier (implicit or explicit) must be the same as one of the authorization ids of the process.
- If the privilege set includes SYSADM authority or SYSCTRL authority, the schema qualifier can be any valid schema name.

If the statement is dynamically prepared, the privilege set is the privileges that are held by the SQL authorization ID of the process unless the process is within a trusted context and the ROLE AS OBJECT OWNER clause is specified. If the process is not running in a trusted context that is defined with the ROLE AS OBJECT OWNER clause and the specified alias name includes a qualifier that is not the same as this authorization ID:

- The privilege set must include SYSADM or SYSCTRL authority.
- The privilege set must include DBADM or DBCTRL authority on the database that contains the table, if the alias is for a table and the value of field DBADM CREATE AUTH on installation panel DSNTIPP is YES.
- The qualifier must be the same as one of the authorization IDs of the process and the privileges that are held by that authorization ID must include the CREATEALIAS privilege. This is an exception to the rule that the privilege set is the privileges that are held by the SQL authorization ID of the process.

Syntax

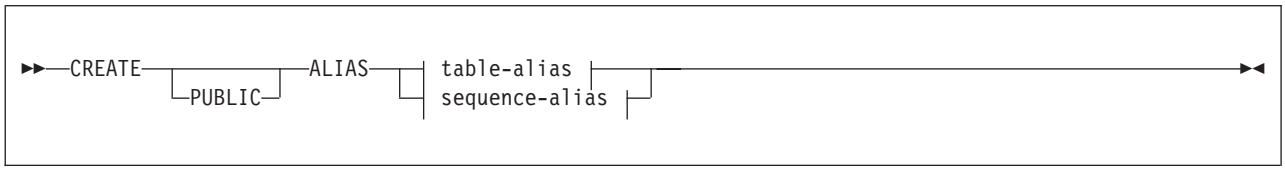
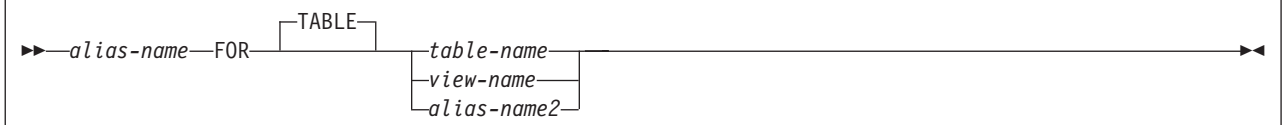


table-alias



sequence-alias

Description

PUBLIC
Specifies that the alias is an object in the system schema SYSPUBLIC.

The PUBLIC keyword is used to create a public alias. If the keyword PUBLIC is not specified, the alias that is created is a private alias.

alias-name
Names the alias.

For a table alias, the name, including the implicit or explicit qualifier, must not identify a table, view, or table alias that exists at the current server, or a table that exists in the SYSIBM.SYSPENDINGOBJECTS catalog table.

For a sequence alias, the name, including the implicit or explicit qualifier, must not identify a sequence or sequence alias that exists at the current server.

If a two-part name is specified, the schema name cannot begin with 'SYS', except if PUBLIC is specified, in which case the schema name must be SYSPUBLIC. The unqualified name must not be the same as an existing synonym.

If the name is qualified, the name can be a two-part or three-part name. If a three-part name is used, the first part must match the value of the field DB2 LOCATION NAME on installation panel DSNTIPR at the current server. (If the current server is not the local DB2, this name is not necessarily the name in the CURRENT SERVER special register.)

When an application uses three-part name aliases for remote objects and DRDA access, the application program must be bound at each location that is specified in the three-part names.

FOR TABLE *table-name*, *view-name*, **or** *alias-name2*
Identifies the table, view, or table alias for which *alias-name* is defined. If another alias name is specified (*alias-name2*), it must not be the same as the new *alias-name* that is being defined (in its fully-qualified form). If a table is identified, it must not be an auxiliary table, a declared temporary table, or a table that is implicitly created for an XML column.

The table or view need not exist at the time the alias is defined. If it does not exist when the alias is created, a warning is returned. However, the referenced object must exist when a SQL statement that contains the alias is used, otherwise an error is returned. If it does exist, the referenced object can be at the current server or at another server. The referenced name must not be an alias that exists at the current server.

FOR SEQUENCE *sequence-name*
Identifies the sequence for which *alias-name* is defined. The *sequence-name* must not be a sequence that is generated by the DB2 subsystem for an identity column or a DOCID column. The schema name must not begin with 'SYS' unless the schema name is 'SYSADM'. *sequence-name* must not be an existing alias for a sequence.

The sequence need not exist at the time the alias is defined. If *sequence-name* does not exist when the alias is created, a warning is returned. However, the referenced object must exist when a SQL statement that contains the alias is used, otherwise an error is returned.

Notes

Owner privileges:
There are no specific privileges on an alias. For more information about ownership of an object, see "Authorization, privileges, permissions, masks, and object ownership" on page 70.

PUBLIC aliases:
If the PUBLIC keyword is specified or if SYSPUBLIC is explicitly specified as the schema qualifier for *alias-name*, a public alias is created.

Resolving an unqualified alias name:
When an unqualified alias name is resolved, private aliases are considered before public aliases.

Conservative binding for aliases for synonyms:
The timestamp for creation of an alias for a synonym must be older than

the timestamp that results from an explicit bind for the package that contains the reference to the alias for the synonym. During the automatic bind process, aliases for synonyms that are created in a release of the DB2 subsystem that is later than the release that was used to explicitly bind the package are not considered for resolution of an alias for a synonym.

Example

Example 1: Create an alias for a catalog table at a DB2 with location name DB2USCALABOA5281.

```
CREATE ALIAS LATABLES FOR DB2USCALABOA5281.SYSIBM.SYSTABLES;
```

Example 2: Create a public alias called SEQS for a sequence named JOE.JOESSEQ.

```
CREATE PUBLIC ALIAS SEQS FOR JOE.JOESSEQ;
```

The alias can be referenced as SYSPUBLIC.SEQS, or simply as SEQS if a private sequence or alias named SEQS does not exist.

CREATE AUXILIARY TABLE

The CREATE AUXILIARY TABLE statement creates an auxiliary table at the current server for storing LOB data.

Invocation

This statement can be embedded in an application program or issued interactively if the value of special register CURRENT RULES is 'DB2' and the table space is explicitly created when the statement is executed. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Do not use this statement if the value of special register CURRENT RULES is 'STD' or if the table space is implicitly created. When the values of the CURRENT RULES special register is 'STD' and a base table is created with LOB columns or altered such that LOB columns are added, DB2 automatically creates the LOB table space, auxiliary table, and index on the auxiliary table for each LOB column. DB2 also automatically creates the LOB table space, auxiliary table, and index on the auxiliary table for each LOB column if the table space is implicitly created. DB2 chooses the names and characteristics of these objects. For more information about the names and the characteristics, see Creating a table with LOB columns.

Authorization

The privilege set that is defined below must include at least one of the following:

- The CREATETAB privilege for the database implicitly or explicitly specified by the IN clause
- DBADM, DBCTRL, or DBMAINT authority for the database
- SYSADM or SYSCTRL authority
- System DBADM

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the application is bound in a trusted context with the ROLE AS OBJECT OWNER clause specifies, a role is the owner. Otherwise, an authorization ID is the owner. If the specified table name includes a qualifier that is not the same as this authorization ID, the privilege set must include SYSADM or SYSCTRL authority, DBADM authority for the database, or DBCTRL authority for the database.

If ROLE AS OBJECT OWNER is in effect, the schema qualifier must be the same as the role, unless the role has the CREATEIN privilege on the schema, SYSADM authority, or SYSCTRL authority.

If ROLE AS OBJECT OWNER is not in effect, one of the following rules applies:

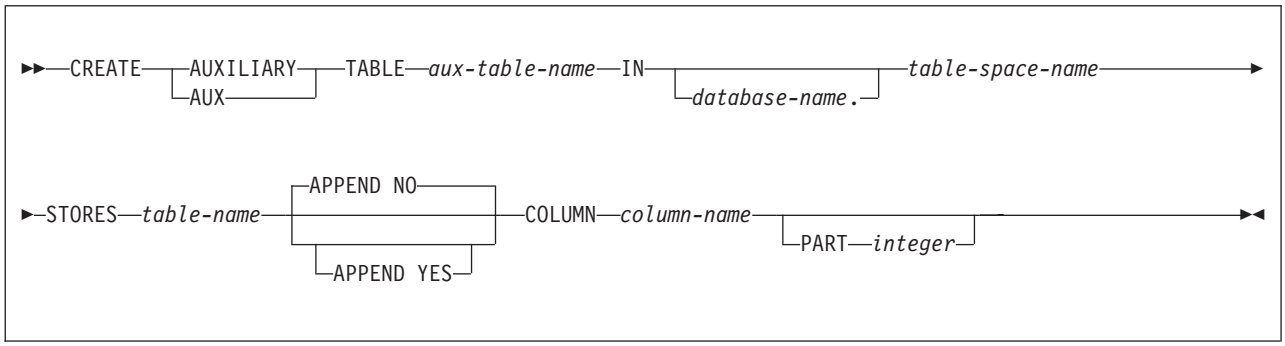
- If the privilege set lacks the CREATIN privilege on the schema, SYSADM authority, or SYSCTRL authority, the schema qualifier (implicit or explicit) must be the same as one of the authorization ids of the process.
- If the privilege set includes SYSADM authority or SYSCTRL authority, the schema qualifier can be any valid schema name.

If the statement is dynamically prepared, the privilege set is the privileges that are held by the SQL authorization ID of the process unless the process is within a trusted context and the ROLE AS OBJECT OWNER clause is specified. In that case,

the privilege set is the set of privileges that are held by the role that is associated with the primary authorization ID of the process. If the process is in a trusted context, any authorization ID can be the qualifier. However, if the process is not in a trusted context and if the specified table name includes a qualifier that is not the same as the SQL authorization ID of the process, the following rules apply:

- If the privilege set includes SYSADM or SYSCTRL authority (or DBADM authority for the database, or DBCTRL authority for the database when creating a table), the schema qualifier can be any valid schema name.
- If the privilege set lacks SYSADM or SYSCTRL authority (or DBADM authority for the database, or DBCTRL authority for the database when creating a table), the schema qualifier is valid only if it is the same as one of the authorization IDs of the process and the privilege set that are held by that authorization ID includes all[26] privileges needed to create the table.

Syntax



Description

AUXILIARY or AUX
Specifies a table that is used to store the LOB data for a LOB column (or a column with a distinct type that is based on a LOB data type).

aux-table-name
Names the auxiliary table. The name, including the implicit or explicit qualifiers, must not identify a table, view, alias, or synonym that exists at the current server, or a table that exists in the SYSIBM.SYSPENDINGOBJECTS catalog table.

IN *database-name.table-space-name* **or IN** *table-space-name*
Identifies the table space in which the auxiliary table is created. The name must identify an empty LOB table space that currently exists at the current server. The LOB table space must be in the same database as the associated base table.

If you specify a database and a table space, the table space must belong to the specified database. If you specify only a table space, it must belong to the database that contains the specified table space. If you specify only a table space, this table space must belong to DSNDB04. This type of table space is only created when SET CURRENT RULES='DB2' is specified.

STORES *table-name* **COLUMN** *column-name*
Identifies the base table and the column of that table that is to be stored in the auxiliary table. If the base table is nonpartitioned, an auxiliary table must not

26. Exception: The CREATETAB privilege is checked on the SQL authorization ID of the process.

already exist for the specified column. If the base table is partitioned, an auxiliary table must not already exist for the specified column and specified partition.

The encoding scheme for the LOB data stored in the auxiliary table is the same as the encoding scheme for the base table. It is either ASCII, EBCDIC, or UNICODE depending on the value of the CCSID clause when the base table was created.

APPEND NO or APPEND YES
Specifies whether append processing is used for the table. The APPEND clause must not be specified for a table in a work file table space.

If the base table is in a range-partitioned table space, the APPEND option on the LOB table might be different for each partition (depending if the LOB table space and associated objects for each partition are created explicitly or implicitly). If the base table is in a partition-by-growth table space, the APPEND attributes of LOB table will be inherited by each partition.

APPEND NO
Specifies that append processing is not used for the table. For insert and LOAD operations, DB2 will attempt to place data rows in a well clustered manner with respect to the value in the row's cluster key columns.

APPEND NO is the default

APPEND YES
Specifies that data rows are placed into the table without regard to clustering during the insert and LOAD operations.

PART *integer*
Specifies the partition of the base table for which the auxiliary table is to store the specified column. You can specify PART only if the base table is defined in a partitioned table space, and no other auxiliary table exists for the same LOB column of the base table.

Notes

Owner privileges: There are no specific privileges on an auxiliary table. For more information about ownership of an object, see "Authorization, privileges, permissions, masks, and object ownership" on page 70.

Determining the number of auxiliary tables to create: If the base table is nonpartitioned, you might need to create one LOB table space and one auxiliary table for each LOB column in the base table. If the base table is partitioned, for each LOB column, you might need to create one LOB table space and one auxiliary table for each partition. For example if your base table has three partitions and two LOB columns, you might need to create three LOB table spaces and three auxiliary tables for each LOB column. In other words, you might need a total of six LOB table spaces and six auxiliary tables.

Auxiliary tables in LOB table spaces that are logged: When you create an auxiliary table in a LOB table space that is LOGGED, and the associated base table space is NOT LOGGED, the logging attribute of the LOB table space is implicitly changed to NOT LOGGED and the logging attributes of the base table space and the LOB table space are linked.

Append processing and unused free space in a table: An update or delete of LOB data creates some free space in the LOB table that can be used by the next insert. If

the table uses append processing, any free space that is not at the end of the table space will not be reused during the insert operation. Any unused free space in the table can be reclaimed by running the REORG utility with either the SHRELEVL REFERENCE or SHRLEVEL CHANGE keywords. The REORG utility is not influenced by the APPEND option.

Example

Assume that a column named EMP_PHOTO with a data type of BLOB(110K) has been added to sample employee table DSN8B10.EMP for each employee's photo. Create auxiliary table EMP_PHOTO_ATAB to store the BLOB data for the BLOB column in LOB table space DSN8D11A.PHOTOLTS.

```
CREATE AUX TABLE EMP_PHOTO_ATAB
   IN DSN8D11A.PHOTOLTS
   STORES DSN8B10.EMP
   COLUMN EMP_PHOTO;
```

CREATE DATABASE

The CREATE DATABASE statement defines a DB2 database at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set that is defined below must include at least one of the following:

- The CREATEDBA privilege
- The CREATEDBC privilege
- SYSADM or SYSCTRL authority
- System DBADM

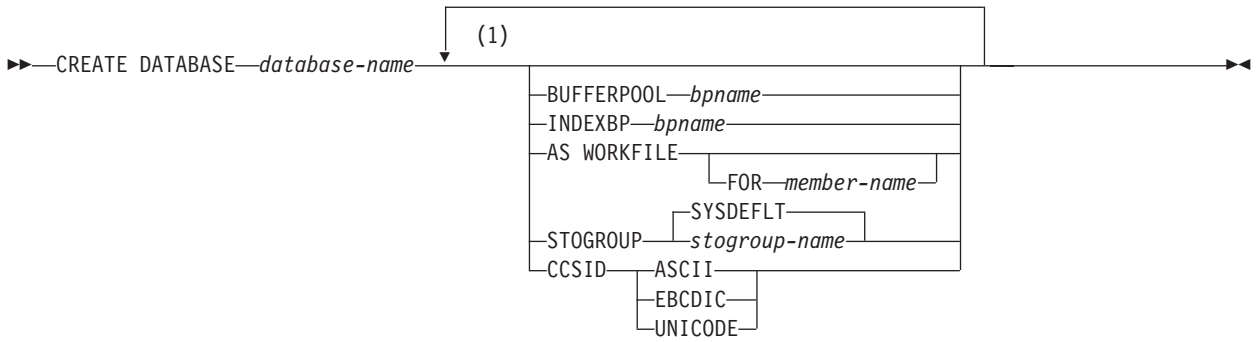
If the database is created as a workfile database, the privilege set that is defined below must include SYSADM authority.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package.

If the statement is dynamically prepared, the privilege set is the privileges that are held by the SQL authorization ID of the process unless the process is within a trusted context and the ROLE AS OBJECT OWNER clause is specified. In that case, the privilege set is the set of privileges that are held by the role that is associated with the primary authorization ID of the process.

See "Notes" on page 1164 for the authorization effect of a successful CREATE DATABASE statement.

Syntax



Notes:

1 The same clause must not be specified more than one time.

Description

database-name

Names the database. The name must not start with DSNDB and must not identify a database that exists at the current server. *database-name* must not be in the form of eight characters that start with DSN followed by exactly five digits. If the database is to be a work file database in a data sharing environment, DSNDB07 is an acceptable work file database name. However, only one member of a data sharing group can use DSNDB07 as the name of its work file database.

BUFFERPOOL *bpname*

Specifies the default buffer pool name to be used for table spaces created within the database. If the database is a work file database, 8KB and 16KB buffer pools cannot be specified. See "Naming conventions" on page 57 for more details about *bpname*.

If you omit the BUFFERPOOL clause, the default 4-KB buffer pool for user data that is specified on installation panel DSNTIP1 is used.

INDEXBP *bpname*

Specifies the default buffer pool name to be used for the indexes created within the database. The name can identify a 4KB, 8KB, 16KB, or 32KB buffer pool. See "Naming conventions" on page 57 for more details about *bpname*.

If you omit the INDEXBP clause, the buffer pool specified for user indexes on installation panel DSNTIP1 is used. The default value for the user indexes field on that panel is BP0.

AS WORKFILE

Specifies the database is a work file database. AS WORKFILE can be specified only in a data sharing environment. Only one work file database can be created for each DB2 subsystem. The work file database is used for work files, created global temporary table, declared temporary tables, and sensitive static scrollable cursors.

PUBLIC implicitly receives the CREATETAB privilege (without GRANT authority) to define a declared temporary table in the work file database. This implicit privilege is not recorded in the DB2 catalog and cannot be revoked.

The CCSID clause is not supported for a work file database. If you specify AS WORKFILE, do not use the CCSID clause.

FOR *member-name*

Specifies the member for which this database is to be created. Specify FOR member-name only in a data sharing environment.

If FOR *member-name* is not specified, the member is the DB2 subsystem on which the CREATE DATABASE statement is executed.

STOGROUP *stogroup-name*

Specifies the storage group to be used, as required, as a default storage group to support DASD space requirements for table spaces and indexes within the database. The default is SYSDEFLT.

CCSID *encoding-scheme*

Specifies the default encoding scheme for data stored in the database. The default applies to table spaces created in the database. All tables stored within a table space must use the same encoding scheme.

ASCII Specifies that the data must be encoded using the ASCII CCSIDs of the server.

EBCDIC
Specifies that the data must be encoded using the EBCDIC CCSIDs of the server.

UNICODE
Specifies that the data must be encoded using the UNICODE CCSIDs of the server.

Usually, each encoding scheme requires only a single CCSID. Additional CCSIDs are needed when mixed, graphic, or UNICODE data is used.

The option defaults to the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

Do not use the CCSID clause if you specify the AS WORKFILE clause.

Notes

If the statement is embedded in an application program, the owner of the plan or package is the owner of the database. If the statement is dynamically prepared, the SQL authorization ID of the process is the owner of the database.

If the owner of the database has the CREATEDBA, SYSADM, or SYSCTRL authority, the owner acquires DBADM authority for the database. DBADM authority for a database includes table privileges on all tables in that database. Thus, if a user with SYSCTRL authority creates a database, that user has table privileges on all tables in that database. This is an exception to the rule that SYSCTRL authority does not include table privileges.

If the owner of the database has the CREATEDBC privilege, but not the CREATEDBA privilege, the owner acquires DBCTRL authority for the database. In this case, no authorization ID has DBADM authority for the database until it is granted by an authorization ID with SYSADM authority.

Examples

Example 1: Create database DSN8D11P. Specify DSN8G110 as the default storage group to be used for the table spaces and indexes in the database. Specify 8KB buffer pool BP8K1 as the default buffer pool to be used for table spaces in the database, and BP2 as the default buffer pool to be used for indexes in the database.

```
CREATE DATABASE DSN8D11P
  STOGROUP DSN8G110
  BUFFERPOOL BP8K1
  INDEXBP BP2;
```

Example 2: Create database DSN8TEMP. Use the defaults for the default storage group and default buffer pool names. Specify ASCII as the default encoding scheme for data stored in the database.

```
CREATE DATABASE DSN8TEMP
  CCSID ASCII;
```

CREATE FUNCTION

The CREATE FUNCTION statement registers a user-defined function with a database server. Each type of function that you can register with this statement is described separately.

External scalar
The function is written in a programming language and returns a scalar value. The external executable routine is registered with a database server along with various attributes of the function. See "CREATE FUNCTION (external scalar)" on page 1166.

External table
The function is written in a programming language. It returns a table to the subselect from which it was started by returning one row at a time, each time that the function is started. The external executable routine is registered with a database server along with various attributes of the function. See "CREATE FUNCTION (external table)" on page 1191.

Sourced
The function is implemented by starting another function (either built-in, external, SQL, or sourced) that exists at the server. The function inherits the attributes of the underlying source function. See "CREATE FUNCTION (sourced)" on page 1210.

SQL scalar
The function is written exclusively in SQL statements and returns a scalar value. The body of an SQL scalar function is written in the SQL procedural language. The function is defined at the current server along with various attributes of the function. See "CREATE FUNCTION (SQL scalar)" on page 1224.

SQL table
The function is written exclusively in SQL statements and returns a set of rows. The body of an SQL table function is written in the SQL procedural language. The function is defined at the current server along with various attributes of the function. See "CREATE FUNCTION (SQL table)" on page 1251.

CREATE FUNCTION (external scalar)

This CREATE FUNCTION statement registers a user-defined external scalar function with a database server. A scalar function returns a single value each time it is invoked.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set defined below must include at least one of the following:

- The CREATEIN privilege on the schema
- SYSADM or SYSCTRL authority
- System DBADM

The authorization ID that matches the schema name implicitly has the CREATEIN privilege on the schema.

If the authorization ID that is used to create the function has installation SYSADM authority, the function is identified as system-defined function.

Additional privileges are required if the function uses a table as a parameter, refers to a distinct type, or is to run in a WLM (workload manager) environment. These privileges are:

- The SELECT privilege on any table that is an input parameter to the function.
- The USAGE privilege on each distinct type that the function references.
- Authority to create programs in the specified WLM environment. This authorization is obtained from an external security product, such as RACF.

At least one of the following additional privileges is required if the SECURED option is specified

- SECADM authority
- CREATE_SECURE_OBJECT privilege

When LANGUAGE is JAVA and a *jar-name* is specified in the EXTERNAL NAME clause, the privilege set must include USAGE on the JAR file.

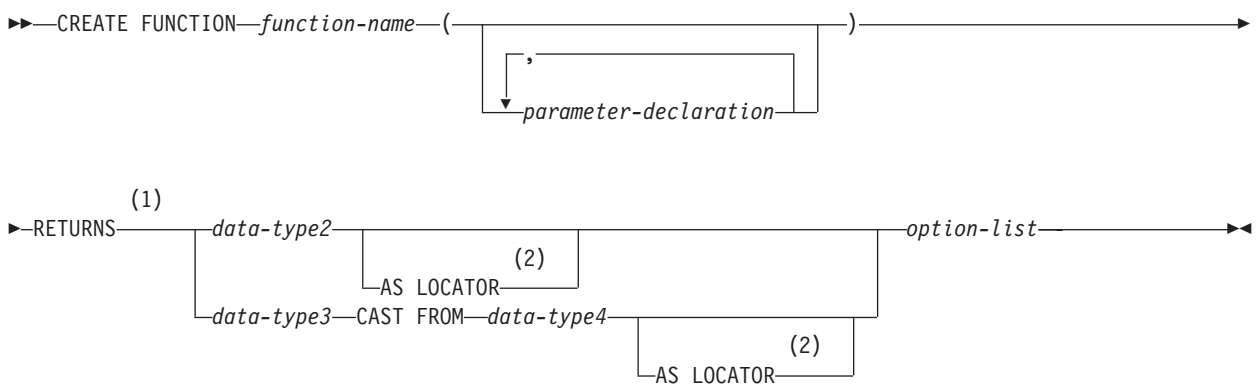
Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the owner is a role, the implicit schema match does not apply and this role needs to include one of the previously listed conditions.

If the statement is dynamically prepared and is not running in a trusted context for which the ROLE AS OBJECT OWNER clause is specified, the privilege set is the set of privileges that are held by the SQL authorization ID of the process. If the schema name is not the same as the SQL authorization ID of the process, one of the following conditions must be met:

- The privilege set includes SYSADM or SYSCTRL authority.

- The SQL authorization ID of the process has the CREATEIN privilege on the schema.

Syntax

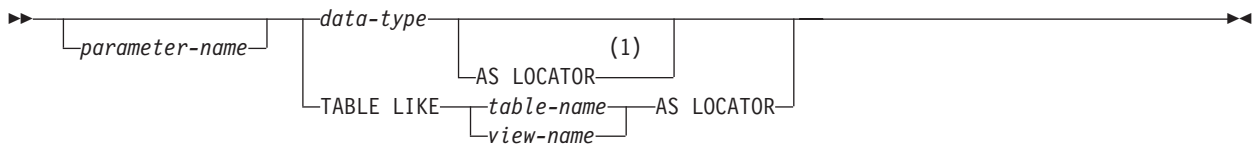


Notes:

1 This clause and the clauses that follow in the *option-list* can be specified in any order.

2 AS LOCATOR can be specified only for a LOB data type or a distinct type based on a LOB data type.

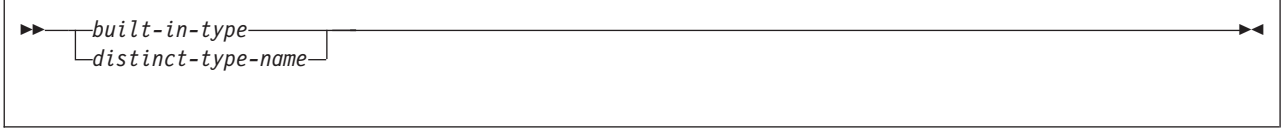
parameter-declaration:



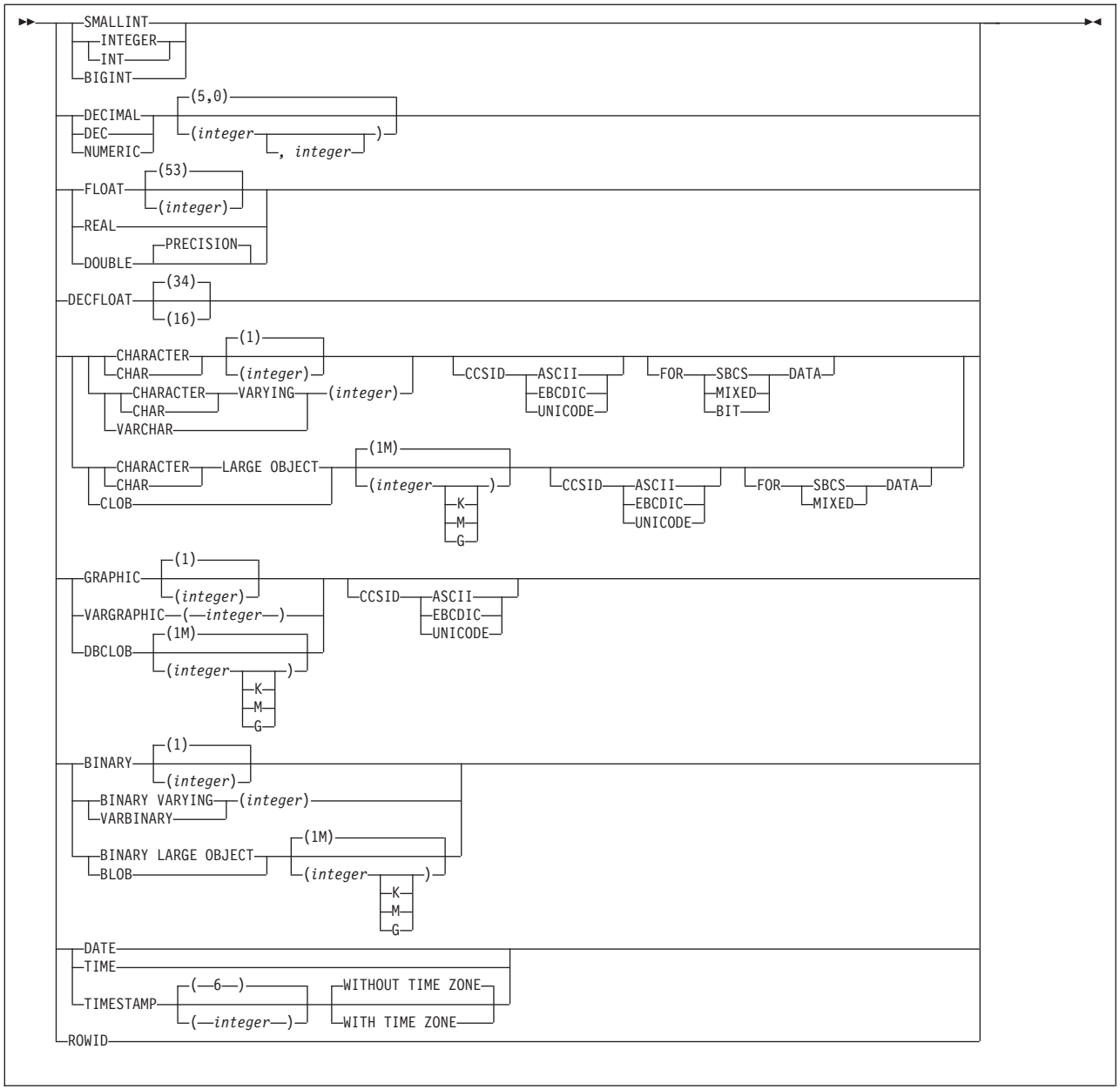
Notes:

1 AS LOCATOR can be specified only for a LOB data type or a distinct type based on a LOB data type.

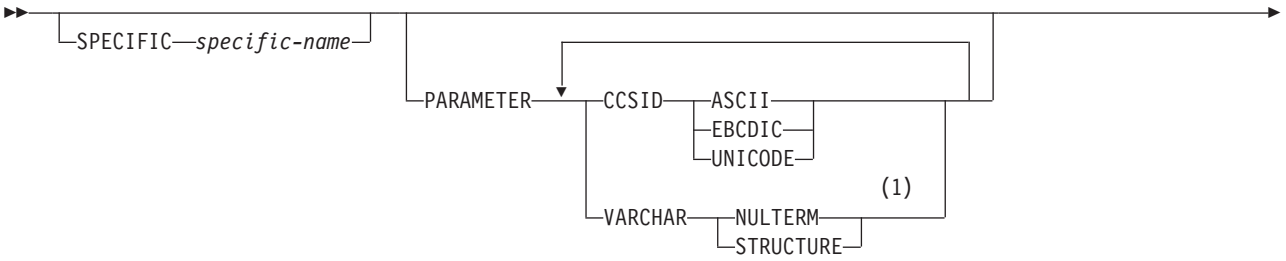
data-type:

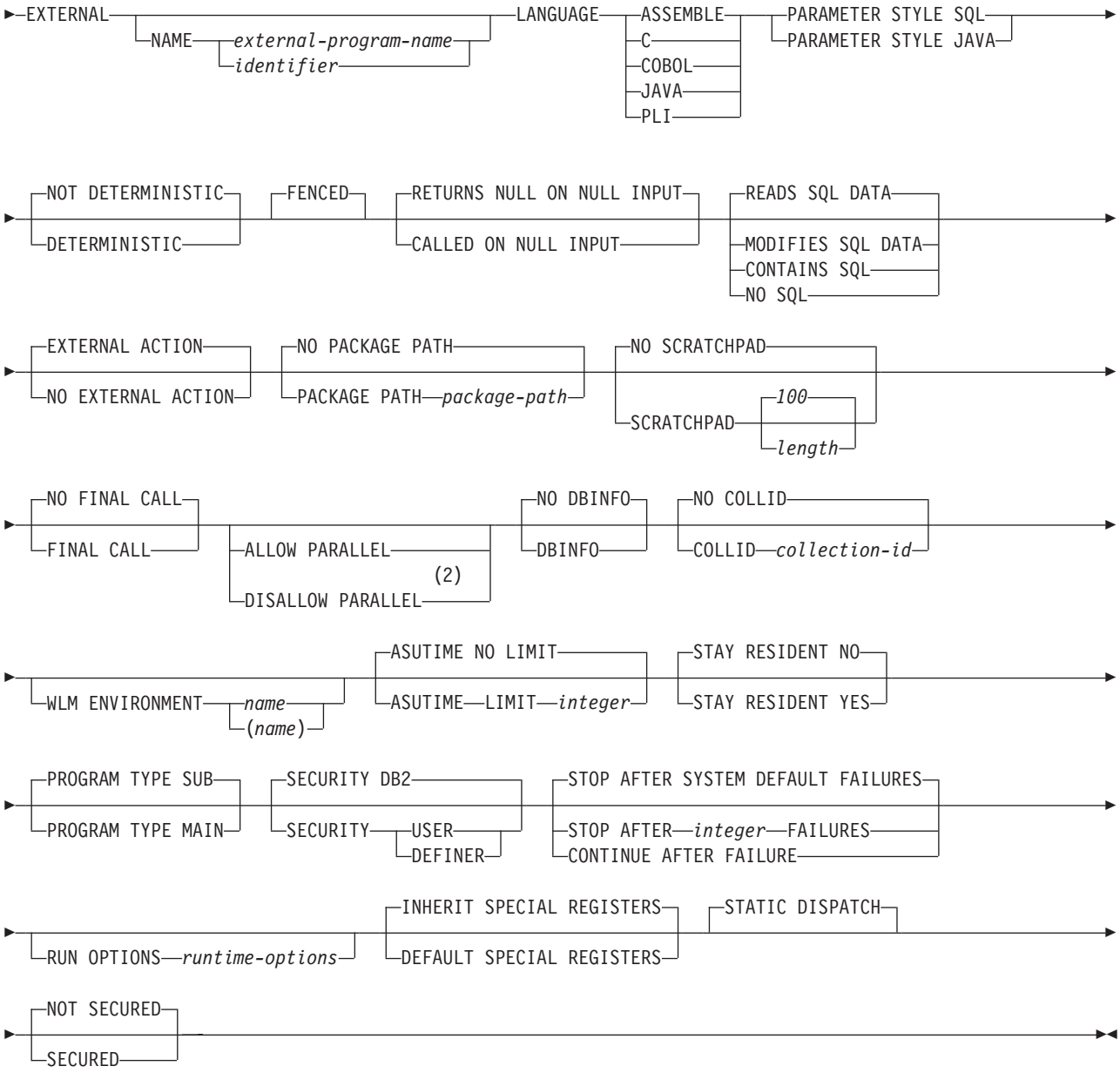


built-in-type:



option-list: (The clauses in the option list can be specified in any order)



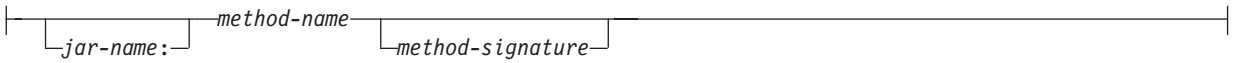


Notes:

1 The same clause must not be specified more than one time.

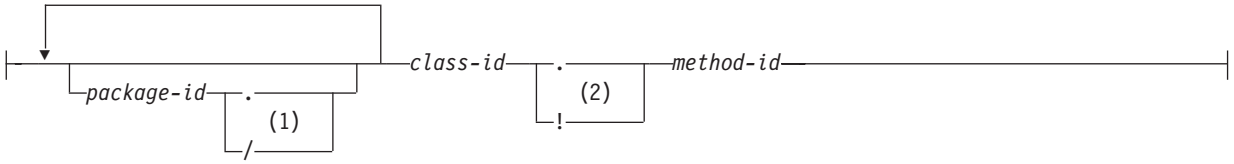
2 If NOT DETERMINISTIC, EXTERNAL ACTION, SCRATCHPAD, or FINAL CALL is specified, DISALLOW PARALLEL is the default.

external-java-routine-name:



jar-name:

method-name:



method-signature:

Notes:

1 The slash (/) is supported for compatibility with previous release of DB2 for z/OS.

2 The exclamation point (!) is supported for compatibility with other products in the DB2 family.

Description

function-name

Names the user-defined function. The name is implicitly or explicitly qualified by a schema name.

The combination of name, schema name, the number of parameters, and the data type of each parameter (without regard for any length, precision, scale, subtype or encoding scheme attributes of the data type) must not identify a user-defined function that exists at the current server. If the function has more than 30 parameters, only the first 30 parameters are used to determine whether the function is unique.

You can use the same name for more than one function if the function signature of each function is unique.

- The unqualified form of *function-name* must not be any of the following system-reserved keywords even if you specify them as delimited identifiers:

```
ALL                 LIKE                UNIQUE
AND                 MATCH               UNKNOWN
ANY                 NOT                 =
BETWEEN             NULL                ¬=
DISTINCT            ONLY                <
```

```
EXCEPT                  OR                      <=
EXISTS                  OVERLAPS                ¬<
FALSE                   SIMILAR                 >
FOR                     SOME                    >=
FROM                    TABLE                   ¬>
IN                      TRUE                    <>
IS                      TYPE
```

The schema name can be 'SYSTOOLS' or 'SYSFUN' if the user who executes the CREATE statement has SYSADM or SYSCTRL privilege. Otherwise, the schema name must not begin with 'SYS' unless the schema name is 'SYSADM'.

(*parameter-declaration***,...)**

Identifies the number of input parameters of the function, and specifies the data type of each parameter. All of the parameters for a function are input parameters and are nullable. There must be one entry in the list for each parameter that the function expects to receive. Although not required, you can give each parameter a name.

A function can have no parameters. In this case, you must code an empty set of parentheses, for example:

```
CREATE FUNCTION WOOFER()
```

parameter-name

Specifies the name of the input parameter. The name is an SQL identifier, and each name in the parameter list must not be the same as any other name.

data-type

Specifies the data type of the input parameter. The data type can be a built-in data type or a distinct type.

built-in-type

The data type of the input parameter is a built-in data type.

For information on the data types, see built-in-type.

For parameters with a character or graphic data type, the PARAMETER CCSID clause or CCSID clause indicates the encoding scheme of the parameter. If you do not specify either of these clauses, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

distinct-type-name

The data type of the input parameter is a distinct type. Any length, precision, scale, subtype, or encoding scheme attributes for the parameter are those of the source type of the distinct type.

If you specify the name of the distinct type without a schema name, DB2 resolves the schema name by searching the schemas in the SQL path.

The implicitly or explicitly specified encoding scheme of all of the parameters with a character or graphic string data type must be the same—either all ASCII, all EBCDIC, or all UNICODE.

Although parameters with a character data type have an implicitly or explicitly specified subtype (BIT, SBCS, or MIXED), the function program can receive character data of any subtype. Therefore, conversion of the input data to the subtype of the parameter might occur when the function

is invoked. An error occurs if mixed data that actually contains DBCS characters is used as the value for an input parameter that is declared with an SBCS subtype.

Parameters with a datetime data type or a distinct type are passed to the function as a different data type:

- A datetime type parameter is passed as a character data type, and the data is passed in ISO format.

  The encoding scheme for a datetime type parameter is the same as the implicitly or explicitly specified encoding scheme of any character or graphic string parameters. If no character or graphic string parameters are passed, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.
- A distinct type parameter is passed as the source type of the distinct type.

AS LOCATOR
Specifies that a locator to the value of the parameter is passed to the function instead of the actual value. Specify AS LOCATOR only for parameters with a LOB data type or a distinct type based on a LOB data type. Passing locators instead of values can result in fewer bytes being passed to the function, especially when the value of the parameter is very large.

The AS LOCATOR clause has no effect on determining whether data types can be promoted, nor does it affect the function signature, which is used in function resolution.

TABLE LIKE *table-name* **or** *view-name* **AS LOCATOR**
Specifies that the parameter is a transition table. However, when the function is invoked, the actual values in the transition table are not passed to the function. A single value is passed instead. This single value is a locator to the table, which the function uses to access the columns of the transition table. A function with a table parameter can only be invoked from the triggered action of a trigger.

The use of TABLE LIKE provides an implicit definition of the transition table. It specifies that the transition table has the same number of columns as the identified table or view. If a table is specified, the transition table includes columns that are defined as implicitly hidden in the table. The columns have the same data type, length, precision, scale, subtype, and encoding scheme as the identified table or view, as they are described in catalog tables SYSCOLUMNS and SYSTABLESPACE. The number of columns and the attributes of those columns are determined at the time the CREATE FUNCTION statement is processed. Any subsequent changes to the number of columns in the table or the attributes of those columns do not affect the parameters of the function.

table-name or *view-name* must identify a table or view that exists at the current server. A view cannot have columns of length 0. The name must not identify a declared temporary table. The table that is identified can contain XML columns; however, the function cannot reference those XML columns. The name does not have to be the same name as the table that is associated with the transition table for the trigger. An unqualified table or view name is implicitly qualified according to the following rules:

- If the CREATE FUNCTION statement is embedded in a program, the implicit qualifier is the authorization ID in the QUALIFIER bind option

when the plan or package was created or last rebound. If QUALIFIER was not used, the implicit qualifier is the owner of the plan or package.

- If the CREATE FUNCTION statement is dynamically prepared, the implicit qualifier is the SQL authorization ID in the CURRENT SCHEMA special register.

When the function is invoked, the corresponding columns of the transition table identified by the table locator and the table or view identified in the TABLE LIKE clause must have the same definition. The data type, length, precision, scale, and encoding scheme of these columns must match exactly. The description of the table or view at the time the CREATE FUNCTION statement was executed is used.

Additionally, a character FOR BIT DATA column of the transition table cannot be passed as input for a table parameter for which the corresponding column of the table specified at the definition is not defined as character FOR BIT DATA. (The definition occurs with the CREATE FUNCTION statement.) Likewise, a character column of the transition table that is not FOR BIT DATA cannot be passed as input for a table parameter for which the corresponding column of the table specified at the definition is defined as character FOR BIT DATA.

For more information about using table locators, see *DB2 Application Programming and SQL Guide*.

RETURNS

Specifies the data type for the result of the function. Consider this clause in conjunction with the optional CAST FROM clause.

data-type2

Specifies the data type of the output. The output parameter is nullable.

The same considerations that apply to the data type and nullability of input parameter, as described under data-type, apply to the data type of the result of the function.

AS LOCATOR

Specifies that the function returns a locator to the value rather than the actual value. You can specify AS LOCATOR only if the output from the function has a LOB data type or a distinct type based on a LOB data type.

data-type3 **CAST FROM** *data-type4*

Specifies the data type of the output of the function (*data-type4*) and the data type in which that output is returned to the invoking statement (*data-type3*). The two data types can be different. For example, for the following definition, the function returns a DOUBLE value, which DB2 converts to a DECIMAL value and then passes to the statement that invoked the function:

```
CREATE FUNCTION SQRT(DECIMAL(15,0))
      RETURNS DECIMAL(15,0) CAST FROM DOUBLE
      ...
```

The value of *data-type4* can be any built-in data type and must be castable to *data-type3*. The value for *data-type3* can be any built-in data type. (For information on casting data types, see "Casting between data types" on page 111.) The encoding scheme of the parameters, if they are string data types, must be the same.

If the PARAMETER VARCHAR clause is specified, *data-type3* and *data-type4* should be specified as VARCHAR.

AS LOCATOR
Specifies that the function returns a locator to the value rather than the value. You can specify AS LOCATOR only if *data-type4* is a LOB data type or a distinct type based on a LOB data type.

SPECIFIC *specific-name*
Specifies a unique name for the function. The name is implicitly or explicitly qualified with a schema name. The name, including the schema name, must not identify the specific name of another function that exists at the current server.

The unqualified form of *specific-name* is an SQL identifier. The qualified form is an SQL identifier (the schema name) followed by a period and an SQL identifier.

If you do not specify a schema name, it is the same as the explicit or implicit schema name of the function name (*function-name*). If you specify a schema name, it must be the same as the explicit or implicit schema name of the function name.

If you do not specify the SPECIFIC clause, the default specific name is the name of the function. However, if the function name does not provide a unique specific name or if the function name is a single asterisk, DB2 generates a specific name in the form of:

```
SQLxxxxxxxxxxxx
```

where `'xxxxxxxxxxxx'` is a string of 12 characters that make the name unique.

The specific name is stored in the SPECIFIC column of the SYSROUTINES catalog table. The specific name can be used to uniquely identify the function in several SQL statements (such as ALTER FUNCTION, COMMENT, DROP, GRANT, and REVOKE) and must be used in DB2 commands (START FUNCTION, STOP FUNCTION, and DISPLAY FUNCTION). However, the function cannot be invoked by its specific name.

PARAMETER CCSID or VARCHAR
Specifies the encoding scheme for character and graphic string parameters, and in the case of LANGUAGE C, specifies that representation of variable length string parameters.

CCSID
Indicates whether the encoding scheme for character and graphic string parameters is ASCII, EBCDIC, or UNICODE. The default encoding scheme is the value specified in the CCSID clauses of the parameter list or RETURNS clause, or in the field DEF ENCODING SCHEME on installation panel DSNTIPF.

This clause provides a convenient way to specify the encoding scheme for *all* string parameters. If individual CCSID clauses are specified for individual parameters in addition to this PARAMETER CCSID clause, the value specified in *all* of the CCSID clauses must be the same value that is specified in this clause.

This clause also specifies the encoding scheme to be used for system-generated parameters of the routine such as message tokens and DBINFO.

VARCHAR
Specifies that the representation of the values of varying length character string-parameters, including, if applicable, the output of the function, for functions which specify LANGUAGE C.

This option can only be specified if LANGUAGE C is also specified.

NULTERM
Specifies that variable length character string parameters are represented in a NUL-terminated string form.

STRUCTURE
Specifies that variable length character string parameters are represented in a VARCHAR structure form.

Using the PARAMETER VARCHAR clause, there is no way to specify the VARCHAR form of an individual parameter as there is with the PARAMETER CCSID clause. The PARAMETER VARCHAR clause only applies to parameters in the parameter list of a function and in the RETURNS clause. It does not apply to system-generated parameters of the routine such as message tokens and DBINFO.

In a data sharing environment, you should not specify the PARAMETER VARCHAR clause until all members of the data sharing group support the clause. If some group members support this clause and others do not, and PARAMETER VARCHAR is specified in an external routine, the routine will encounter different parameter forms depending on which group member invokes the routine.

EXTERNAL
Specifies that the CREATE FUNCTION statement is being used to define a new function that is based on code that is written in an external programming language.

DB2 loads the load module when the function is invoked. The load module is created when the program that contains the function body is compiled and link-edited. The load module does not need to exist when the CREATE FUNCTION statement is executed. However, it must exist and be accessible by the current server when the function is invoked.

You can specify the EXTERNAL clause in one of the following ways:

```
EXTERNAL
EXTERNAL NAME PKJVSP1
EXTERNAL NAME 'PKJVSP1'
```

If you specify an external program name, you must use the NAME keyword. For example, this syntax is not valid:

```
EXTERNAL PKJVSP1
```

NAME *external-program-name* **or** *identifier*
Identifies the user-written code that implements the user-defined function.

If LANGUAGE is JAVA, *external-program-name* must be specified and enclosed in single quotation marks, with no extraneous blanks within the single quotation marks. It must specify a valid *external-java-routine-name*. If multiple *external-program-name*s are specified, the total length of all of them must not be greater than 1305 bytes and they must be separated by a space or a line break. Do not specify a JAR file for a JAVA function for which NO SQL is also specified.

An *external-java-routine-name* contains the following parts:

jar-name
Identifies the name given to the JAR file when it was installed in the database. The name contains *jar-id*, which can optionally be qualified with a schema. Examples are "myJar" and "mySchema.myJar." The unqualified *jar-id* is implicitly qualified with a schema name according to the following rules:

- If the statement is embedded in a program, the schema name is the authorization ID in the QUALIFIER bind option when the package or plan was created or last rebound. If the QUALIFIER was not specified, the schema name is the owner of the package or plan.
- If the statement is dynamically prepared, the schema name is the SQL authorization ID in the CURRENT SCHEMA special register.

If *jar-name* is specified, it must exist when the CREATE FUNCTION statement is processed.

If *jar-name* is not specified, the function is loaded from the class file directly instead of being loaded from a JAR file. DB2 searches the directories in the CLASSPATH associated with the WLM Environment. Environmental variables for Java routines are specified in a data set identified in a JAVAENV DD card on the JCL used to start the address space for a WLM-managed function.

method-name
Identifies the name of the method and must not be longer than 254 bytes. Its package, class, and method ID's are specific to Java and as such are not limited to 18 bytes. In addition, the rules for what these can contain are not necessarily the same as the rules for an SQL ordinary identifier.

package-id
Identifies a package. The concatenated list of *package-ids* identifies the package that the class identifier is part of. If the class is part of a package, the method name must include the complete package prefix, such as "myPacks.UserFuncs." The Java virtual machine looks in the directory "/myPacks/UserFuncs/" for the classes.

class-id
Identifies the class identifier of the Java object.

method-id
Identifies the method identifier with the Java class to be invoked.

method-signature
Identifies a list of zero or more Java data types for the parameter list and must not be longer than 1024 bytes. Specify the *method-signature* if the user-defined function involves any input or output parameters that can be NULL. When the function being created is called, DB2 searches for a Java method with the exact *method-signature*. The number of *java-datatype* elements specified indicates how many parameters that the Java method must have.

A Java procedure can have no parameters. In this case, you code an empty set of parentheses for *method-signature*. If a Java *method-signature* is not specified, DB2 searches for a Java method with a signature derived from the default JDBC types associated with the SQL types specified in the parameter list of the CREATE FUNCTION statement.

For other values of LANGUAGE, the name can be a string constant that is no longer than 8 characters. It must conform to the naming conventions for

load modules. Alphabetical extenders for national languages can be used as the first character and as subsequent characters in the load module name.

If you do not specify the NAME clause, 'NAME *function-name*' is implicit. In this case, *function-name* must not be longer than 8 characters.

LANGUAGE
Specifies the language interface convention to which the body of the function is written. All programs must be designed to run in IBM's Language Environment environment.

ASSEMBLE
The function is written in Assembler.

C The function is written in C or C++.

COBOL
The function is written in COBOL, including the object-oriented language extensions.

JAVA
The user-defined function is written in Java and is executed in the Java Virtual Machine. When LANGUAGE JAVA is specified, the EXTERNAL NAME clause must also be specified with a valid *external-java-routine-name* and PARAMETER STYLE must be specified with JAVA.

Do not specify LANGUAGE JAVA when SCRATCHPAD, FINAL CALL, DBINFO, PROGRAM TYPE MAIN, or RUN OPTIONS is in effect.

PLI
The function is written in PL/I.

PARAMETER STYLE
Specifies the conventions for passing parameters to and returning a value from the function.

SQL
Specifies the parameter passing convention that supports passing null values both as input and for output. The parameters that are passed between the invoking SQL statement and the function include:

- *n* parameters for the input parameters that are specified for the function
- A parameter for the result of the function
- *n* parameters for the indicator variables for the input parameters
- A parameter for the indicator variable for the result
- The SQLSTATE to be returned to DB2
- The qualified name of the function
- The specific name of the function
- The SQL diagnostic string to be returned to DB2
- The function can also pass from zero to three additional parameters:
  - The scratchpad, if SCRATCHPAD is specified
  - The call type, if FINAL CALL is specified
  - The DBINFO structure, if DBINFO is specified

JAVA
Indicates that the user-defined function uses a convention for passing parameters that conforms to the Java and SQLJ specifications.

PARAMETER STYLE JAVA can be specified only if LANGUAGE is specified as JAVA. JAVA must be specified for PARAMETER STYLE when LANGUAGE JAVA is specified.

NOT DETERMINISTIC or DETERMINISTIC
Specifies whether the function returns the same results each time that the function is invoked with the same input arguments.

NOT DETERMINISTIC
The function might not return the same result each time that the function is invoked with the same input arguments. The function depends on some state values that affect the results. DB2 uses this information to disable the merging of views and table expressions when processing SELECT or SQL data change statements that refer to this function. An example of a function that is not deterministic is one that generates random numbers, or any function that contains SQL statements.

NOT DETERMINISTIC is the default.

Some functions that are not deterministic can receive incorrect results if the function is executed by parallel tasks. Specify the DISALLOW PARALLEL clause for these functions.

DETERMINISTIC
The function always returns the same result each time that the function is invoked with the same input arguments. An example of a deterministic function is a function that calculates the square root of the input. DB2 uses this information to enable the merging of views and table expressions for SELECT or SQL data change statements that refer to this function. DETERMINISTIC is not the default. If applicable, specify DETERMINISTIC to prevent non-optimal access paths from being chosen for SQL statements that refer to this function.

DB2 does not verify that the function program is consistent with the specification of DETERMINISTIC or NOT DETERMINISTIC.

FENCED
Specifies that the external function runs in an external address space to prevent the function from corrupting DB2 storage.

FENCED is the default.

RETURNS NULL ON NULL INPUT or CALLED ON NULL INPUT
Specifies whether the function is called if any of the input arguments is null at execution time.

RETURNS NULL ON NULL INPUT
The function is not called if any of the input arguments is null. The result is the null value. RETURNS NULL ON INPUT is the default.

CALLED ON NULL INPUT
The function is called regardless of whether any of the input arguments are null, making the function responsible for testing for null argument values. The function can return a null or nonnull value.

MODIFIES SQL DATA, READS SQL DATA, CONTAINS SQL, or NO SQL
Specifies the classification of SQL statements that the function can execute. DB2 verifies that the SQL statements that the function issues are consistent with this specification. For the data access classification of each statement, see Table 162 on page 2030.

MODIFIES SQL DATA
Specifies that the function can execute any SQL statement except the statements that are not supported in functions. Do not specify MODIFIES SQL DATA when ALLOW PARALLEL is in effect.

READS SQL DATA
Specifies that the function can execute statements with a data access indication of READS SQL DATA, CONTAINS SQL, or NO SQL. The function cannot execute SQL statements that modify data. The default is **READS SQL DATA**.

CONTAINS SQL
Specifies that the function can execute only SQL statements with an access indication of CONTAINS SQL or NO SQL. The function cannot execute statements that read or modify data.

NO SQL
Specifies that the function can execute only SQL statements with a data access classification of NO SQL. Do not specify NO SQL for a JAVA function that uses a JAR file.

EXTERNAL ACTION or NO EXTERNAL ACTION
Specifies whether the function takes an action that changes the state of an object that DB2 does not manage. An example of an external action is sending a message or writing a record to a file.

Because DB2 uses the RRS attachment for external functions, DB2 can participate in two-phase commit with any other resource manager that uses RRS. For resource managers that do not use RRS, there is no coordination of commit or rollback operations on non-DB2 resources.

EXTERNAL ACTION
Specifies that the function can take an action that changes the state of an object that DB2 does not manage.

Some SQL statements that invoke functions with external actions can result in incorrect results if parallel tasks execute the function. For example, if the function sends a note for each initial call to it, one note is sent for each parallel task instead of once for the function. Specify the DISALLOW PARALLEL clause for functions that do not work correctly with parallelism.

If you specify EXTERNAL ACTION, DB2:

- Materializes the views and table expressions in SELECT or data change statements statements that refer to the function. This materialization can adversely affect the access paths that are chosen for the SQL statements that refer to this function. Do not specify EXTERNAL ACTION if the function does not have an external action.
- Does not move the function from one task control block (TCB) to another between FETCH operations.
- Does not allow another function or stored procedure to use the TCB until the cursor is closed. This is also applicable for cursors declared WITH HOLD.

The only changes to resources made outside of DB2 that are under the control of commit and rollback operations are those changes made under RRS control.

EXTERNAL ACTION is the default.

NO EXTERNAL ACTION
Specifies that the function does not take any action that changes the state of an object that DB2 does not manage. DB2 uses this information to enable the merging of views and table expressions for SELECT and data change statements that refer to this function. If applicable, specify NO EXTERNAL ACTION to prevent non-optimal access paths from being chosen for SQL statements that refer to this function.

Although the scope of global variables are beyond the scope of the routine, global variables can be set in the routine body when NO EXTERNAL ACTION is specified.

DB2 does not verify that the function program is consistent with the specification of EXTERNAL ACTION or NO EXTERNAL ACTION.

NO PACKAGE PATH or PACKAGE PATH *package-path*
Specifies the package path to use when the function is run. This is the list of the possible package collections into which the DBRM this is associated with the function is bound.

NO PACKAGE PATH
Specifies that the list of package collections for the function is the same as the list of package collection IDs for the program that invokes the function. If the program that invokes the function does not use a package, DB2 resolves the package by using the CURRENT PACKAGE PATH special register, the CURRENT PACKAGESET special register, or the PKLIST bind option (in this order). For information about how DB2 uses these three items, see *DB2 Application Programming and SQL Guide*.

PACKAGE PATH *package-path*
Specifies a list of package collections, in the same format as the SET CURRENT PACKAGE PATH special register.

If the COLLID clause is specified with PACKAGE PATH, the COLLID clause is ignored when the function is invoked.

The *package-path* value that is provided when the function is created is checked when the function is invoked. If *package-path* contains SESSION_USER (or USER), PATH, or PACKAGE PATH, an error is returned when the *package-path* value is checked.

NO SCRATCHPAD or SCRATCHPAD
Specifies whether DB2 is to provide a scratchpad for the function. It is strongly recommended that external functions be reentrant, and a scratchpad provides an area for the function to save information from one invocation to the next.

NO SCRATCHPAD
Specifies that a scratchpad is not allocated and passed to the function. NO SCRATCHPAD is the default.

SCRATCHPAD *length*
Specifies that when the function is invoked for the first time, DB2 allocates memory for a scratchpad. A scratchpad has the following characteristics:

- *length* must be between 1 and 32767. The default value is 100 bytes.
- DB2 initializes the scratchpad to all binary zeros (X'00''s).
- The scope of a scratchpad is the SQL statement. For each reference to the function in an SQL statement, there is one scratchpad. For example, assuming that function UDFX was defined with the SCRATCHPAD keyword, three scratchpads are allocated for the three references to UDFX in the following SQL statement:

```
SELECT A, UDFX(A) FROM TABLEB
 WHERE UDFX(A) > 103 OR UDFX(A) < 19;
```

If the function is run under parallel tasks, one scratchpad is allocated for each parallel task of each reference to the function in the SQL statement. This can lead to unpredictable results. For example, if a function uses the scratchpad to count the number of times that it is invoked, the count reflects the number of invocations done by the parallel task and not the SQL statement. Specify the DISALLOW PARALLEL clause for functions that will not work correctly with parallelism.

- The scratchpad is persistent. DB2 preserves its content from one invocation of the function to the next. Any changes that the function makes to the scratchpad on one call are still there on the next call. DB2 initializes the scratchpads when it begins to execute an SQL statement. DB2 does not reset scratchpads when a correlated subquery begins to execute.
- The scratchpad can be a central point for the system resources that the function acquires. If the function acquires system resources, specify FINAL CALL to ensure that DB2 calls the function one more time so that the function can free those system resources.

Each time the function invoked, DB2 passes an additional argument to the function that contains the address of the scratchpad.

If you specify SCRATCHPAD, DB2:

- Does not move the function from one task control block (TCB) to another between FETCH operations.
- Does not allow another function or stored procedure to use the TCB until the cursor is closed. This is also applicable for cursors declared WITH HOLD.

Do not specify SCRATCHPAD when LANGUAGE JAVA is in effect.

NO FINAL CALL or FINAL CALL

Specifies whether a *final call* is made to the function. A final call enables the function to free any system resources that it has acquired. A final call is useful when the function has been defined with the SCRATCHPAD keyword and the function acquires system resource and anchors them in the scratchpad.

NO FINAL CALL

Specifies that a final call is not made to the function. The function does not receive an additional argument that specifies the type of call. NO FINAL CALL is the default.

FINAL CALL

Specifies that a final call is made to the function. To differentiate between final calls and other calls, the function receives an additional argument that specifies the type of call. The types of calls are:

First call

Specifies that the first call to the function for this reference to the function in this SQL statement. A first call is a normal call—SQL arguments are passed and the function is expected to return a result.

Normal call

Specifies that SQL arguments are passed and the function is expected to return a result.

Final call

Specifies that the last call to the function to enable the function to free resources. A final call is not a normal call. If an error occurs, DB2 attempts to make the final call unless the function abended. A final call occurs at these times:

- *End of statement:* When the cursor is closed for cursor-oriented statements, or the execution of the statement has completed.
- *End of a parallel task:* When the function is executed by parallel tasks.
- *End of transaction:* When normal end of statement processing does not occur. For example, the logic of an application, for some reason, bypasses closing the cursor.

If a commit operation occurs while a cursor defined as WITH HOLD is open, a final call is made when the cursor is closed or the application ends. If a commit occurs at the end of a parallel task, a final call is made regardless of whether a cursor defined as WITH HOLD is open.

If a commit, rollback, or abort operation causes the final call, the function cannot issue any SQL statements when it is invoked.

Some functions that use a final call can receive incorrect results if parallel tasks execute the function. For example, if a function sends a note for each final call to it, one note is sent for each parallel task instead of once for the function. Specify the DISALLOW PARALLEL clause for functions that have inappropriate actions when executed in parallel.

Do not specify FINAL CALL when LANGUAGE JAVA is in effect.

ALLOW or DISALLOW PARALLEL

For a single reference to the function, specifies whether parallelism can be used when the function is invoked. Although parallelism can be used for most scalar functions, some functions such as those that depend on a single copy of the scratchpad cannot be invoked with parallel tasks.

Consider these characteristics when determining which clause to use:

- If all invocations of the function are completely independent from one another, specify ALLOW PARALLEL.
- If each invocation of the function updates the scratchpad, providing values that are of interest to the next invocation, such as incrementing a counter, specify DISALLOW PARALLEL.
- If the scratchpad is used only so that some expensive initialization processing is performed a minimal number of times, specify ALLOW PARALLEL.
- If the function performs some external action that should apply to only one partition, specify DISALLOW PARALLEL.
- If the function is defined with MODIFIES SQL DATA, specify DISALLOW PARALLEL, not ALLOW PARALLEL.

ALLOW PARALLEL is the default unless NOT DETERMINISTIC, EXTERNAL ACTION, SCRATCHPAD, or FINAL CALL is specified, in which case, DISALLOW PARALLEL is the default.

ALLOW PARALLEL

Specifies that DB2 can consider parallelism for the function. Parallelism is not forced on the SQL statement that invokes the function or on any SQL statement in the function. Existing restrictions on parallelism apply.

DISALLOW PARALLEL
Specifies that DB2 does not consider parallelism for the function.

NO DBINFO or DBINFO
Specifies whether additional status information is passed to the function when it is invoked.

NO DBINFO
No additional information is passed. NO DBINFO is the default.

DBINFO
An additional argument is passed when the function is invoked. The argument is a structure that contains information such as the application runtime authorization ID, the schema name, the name of a table or column that the function might be inserting into or updating, and identification of the database server that invoked the function. For details about the argument and its structure, see *DB2 Application Programming and SQL Guide*.

Do not specify DBINFO when LANGUAGE JAVA is in effect.

NO COLLID or COLLID *collection-id*
Identifies the package collection that is to be used when the function is executed. This is the package collection into which the DBRM that is associated with the function program is bound.

NO COLLID
The package collection for the function is the same as the package collection of the program that invokes the function. If a trigger invokes the function, the collection of the trigger package is used. If the invoking program does not use a package, DB2 resolves the package by using the CURRENT PACKAGE PATH special register, the CURRENT PACKAGESET special register, or the PKLIST bind option (in this order). For details about how DB2 uses these three items, see the information on package resolution in *DB2 Application Programming and SQL Guide*.

NO COLLID is the default.

COLLID *collection-id*
The name of the package collection that is to be used when the function is executed.

WLM ENVIRONMENT
Identifies the WLM (workload manager) application environment in which the function is to run. The *name* of the WLM environment is an SQL identifier.

If you do not specify WLM ENVIRONMENT, the function runs in the WLM-established stored procedure address space that is specified at installation time. When LANGUAGE is JAVA, you must specify WLM ENVIRONMENT, and the WLM environment in which the function is to run must be Java-enabled.

name
The WLM environment in which the function must run. If another user-defined function or a stored procedure calls the function and that calling routine is running in an address space that is not associated with the WLM environment, DB2 routes the function request to a different address space.

(name,*)
When an SQL application program directly invokes the function, the WLM environment in which the function runs.

If another user-defined function or a stored procedure calls the function, the function runs in same environment that the calling routine uses. In this case, authorization to run the function in the WLM environment is not checked because the authorization of the calling routine suffices.

Users must have the appropriate authorization to execute functions in the specified WLM environment. For an example of a RACF command that provides this authorization, see Running external functions in WLM environments.

ASUTIME
Specifies the total amount of processor time, in CPU service units, that a single invocation of the function can run. The value is unrelated to the ASUTIME column of the resource limit specification table. This option is ignored if LANGUAGE JAVA is specified.

When you are debugging a function, setting a limit can be helpful if the function gets caught in a loop. For information on service units, see *z/OS MVS Initialization and Tuning Guide*.

NO LIMIT
There is no limit on the service units. NO LIMIT is the default.

LIMIT *integer*
The limit on the number of CPU service units is a positive *integer* in the range of 1 to 2 147 483 647. If the procedure uses more service units than the specified value, DB2 cancels the procedure. The CPU cycles that are consumed by parallel tasks in a procedure do not contribute towards the specified ASUTIME LIMIT.

STAY RESIDENT
Specifies whether the load module for the function is to remain resident in memory when the function ends. This option is ignored if LANGUAGE JAVA is specified.

NO The load module is deleted from memory after the function ends. Use NO for non-reentrant functions. NO is the default.

YES
The load module remains resident in memory after the function ends. Use YES for reentrant functions.

PROGRAM TYPE
Specifies whether the function program runs as a main routine or a subroutine.

SUB
The function runs as a subroutine. With LANGUAGE JAVA, PROGRAM TYPE SUB is the only valid option. SUB is the default.

MAIN
The function runs as a main routine.

SECURITY
Specifies how the function interacts with an external security product, such as RACF, to control access to non-SQL resources.

DB2
The function does not require an external security environment. If the function accesses resources that an external security product protects, the access is performed using the authorization ID that is associated with the WLM-established stored procedure address space.

DB2 is the default.

USER
An external security environment should be established for the function. If the function accesses resources that the external security product protects, the access is performed using the primary authorization ID of the process that invoked the function.

DEFINER
An external security environment should be established for the function. If the function accesses resources that the external security product protects, the access is performed using the authorization ID of the owner of the function.

STOP AFTER SYSTEM DEFAULT FAILURES, STOP AFTER *nn* **FAILURES, or CONTINUE AFTER FAILURE**
Specifies whether the routine is to be put in a stopped state after some number of failures.

STOP AFTER SYSTEM DEFAULT FAILURES
Specifies that this routine should be placed in a stopped state after the number of failures indicated by the value of field MAX ABEND COUNT on installation panel DSNTIPX. This is the default.

STOP AFTER *nn* **FAILURES**
Specifies that this routine should be placed in a stopped state after *nn* failures. The value *nn* can be an integer from 1 to 32767.

CONTINUE AFTER FAILURE
Specifies that this routine should not be placed in a stopped state after any failure.

RUN OPTIONS *runtime-options*
Specifies the Language Environment runtime options to be used for the function. You must specify *runtime-options* as a character string that is no longer than 254 bytes. If you do not specify RUN OPTIONS or pass an empty string, DB2 does not pass any runtime options to Language Environment, and Language Environment uses its installation defaults. For a description of the Language Environment runtime options, see *z/OS Language Environment Programming Reference*.

Do not specify RUN OPTIONS when LANGUAGE JAVA is in effect.

INHERIT SPECIAL REGISTERS or DEFAULT SPECIAL REGISTERS
Specifies how special registers are set on entry to the routine.

INHERIT SPECIAL REGISTERS
Specifies that the values of special registers are inherited according to the rules listed in the table for characteristics of special registers in a user-defined function in Table 40 on page 205.

DEFAULT SPECIAL REGISTERS
Specifies that special registers are initialized to the default values, as indicated by the rules in the table for characteristics of special registers in a user-defined function in Table 40 on page 205.

STATIC DISPATCH
At function resolution time, DB2 chooses a function based on the static (or declared) types of the function parameters. STATIC DISPATCH is the default.

NOT SECURED or SECURED

Specifies if the function is considered secure for row access control and column access control.

NOT SECURED

Specifies that the function is not considered as secure for row access control and column access control.

NOT SECURED is the default.

When the function is invoked, the arguments of the function must not reference a column for which a column mask is enabled when the table is using active column access control.

SECURED

Specifies that the function is considered secure for row access control and column access control.

The function must be defined with SECURED when it is referenced in a row permission or a column mask.

Notes

Owner privileges:

The owner is authorized to execute the function (EXECUTE privilege) and has the ability to grant these privileges to others. For more information, see "GRANT (function or procedure privileges)" on page 1703. For more information about ownership of the object, see "Authorization, privileges, permissions, masks, and object ownership" on page 70.

Choosing data types for parameters:

When you choose the data types of the input and output parameters for your function, consider the rules of promotion that can affect the values of the parameters. (See "Promotion of data types" on page 110). For example, a constant that is one of the input arguments to the function might have a built-in data type that is different from the data type that the function expects, and more significantly, might not be promotable to that expected data type. Based on the rules of promotion, consider using the following data types for parameters:

- INTEGER instead of SMALLINT
- DOUBLE instead of REAL
- VARCHAR instead of CHAR
- VARGRAPHIC instead of GRAPHIC
- VARBINARY instead of BINARY

For portability of functions across platforms that are not DB2 for z/OS, do not use the following data types, which might have different representations on different platforms:

- FLOAT. Use DOUBLE or REAL instead.
- NUMERIC. Use DECIMAL instead.

Specifying the encoding scheme for parameters:

The implicitly or explicitly specified encoding scheme of all of the parameters with a character or graphic string data type (both input and output parameters) must be the same—either all ASCII, all EBCDIC, or all UNICODE.

Determining the uniqueness of functions in a schema:

At the current server, the function signature of each function, which is the

qualified function name combined with the number and data types of the input parameters, must be unique. If the function has more than 30 input parameters, only the data types of the first 30 are used to determine uniqueness. This means that two different schemas can each contain a function with the same name that have the same data types for all of their corresponding data types. However, a single schema must not contain multiple functions with the same name that have the same data types for all of their corresponding data types.

When determining whether corresponding data types match, DB2 does not consider any length, precision, or scale attributes in the comparison. DB2 considers the synonyms of data types as a match. For example, REAL and FLOAT, and DOUBLE and FLOAT are considered a match. Therefore, CHAR(8) and CHAR(35) are considered to be the same, as are DECIMAL(11,2), DECIMAL(4,3), DECFLOAT(16) and DECFLOAT(34), TIMESTAMP(6) and TIMESTAMP(9), TIMESTAMP(6) WITH TIME ZONE and TIMESTAMP(9) WITH TIME ZONE. Furthermore, the character and graphic types, and the timestamp types are considered to be the same. For example, the following are considered to be the same type: CHAR and GRAPHIC, VARCHAR and VARGRAPHIC, CLOB and DBCLOB, TIMESTAMP WITHOUT TIME ZONE and TIMESTAMP WITH TIME ZONE. CHAR(13) and GRAPHIC(8) are considered to be the same type. An error is returned if the signature of the function being created is a duplicate of a signature for an existing user-defined function with the same name and schema.

Assume that the following statements are executed to create four functions in the same schema. The second and fourth statements fail because they create functions that are duplicates of the functions that the first and third statements created.

```
CREATE FUNCTION PART (INT, CHAR(15)) ...
CREATE FUNCTION PART (INTEGER, CHAR(40)) ...
CREATE FUNCTION ANGLE (DECIMAL(12,2)) ...
CREATE FUNCTION ANGLE (DEC(10,7)) ...
```

Character string representation considerations:

The PARAMETER VARCHAR clause is specific to LANGUAGE C functions because of the native use of NUL-terminated strings in C. VARCHAR structure representation is useful when character string data is known to contain embedded NUL-terminators. It is also useful when it cannot be guaranteed that character string data does not contain embedded NUL-terminators.

PARAMETER VARCHAR does not apply to fixed length character strings, VARCHAR FOR BIT DATA, CLOB, DBCLOB, or implicitly generated parameters. The clause does not apply to VARCHAR FOR BIT DATA because BIT DATA can contain X'00' characters, and its value representation starts with length information. It does not apply to LOB data because a LOB value representation starts with length information.

PARAMETER VARCHAR does not apply to optional parameters that are implicitly provided to an external function. For example, a CREATE FUNCTION statement for LANGUAGE C must also specify PARAMETER STYLE SQL, which returns an SQLSTATE NUL-terminated character string; that SQLSTATE will not be represented in VARCHAR structured form. Likewise, none of the parameters that represent the qualified name of the function, the specific name of the function, or the SQL diagnostic string that is returned to the database manager will be represented in VARCHAR structured form.

Considerations for accessing message tokens and DBINFO:

DB2 returns system-generated parameters from a routine, such as message tokens and DBINFO. The message tokens and DBINFO are character string data. The CCSID for system-generated string parameters is determined from the CCSID that is in effect for string parameters that are defined for the routine. If the parameter list for the routine does not include any character or graphic string parameters, the CCSID for system-generated string parameters is determined from the PARAMETER CCSID option that is in effect for the routine. For example, with a Unicode database, you can specify PARAMETER CCSID EBCDIC to have the system-generated string parameters returned to the invoking application in EBCDIC.

Overriding a built-in function:

Giving a function the same name as a built-in function is not a recommended practice unless you are trying to change the functionality of the built-in function.

If you do intend to create a function with the same name as a built-in function, be careful to maintain the uniqueness of its function signature. If your function has the same name and data types of the corresponding parameters of the built-in function but implements different logic, DB2 might choose the wrong function when the function is invoked with an unqualified function name. Thus, the application might fail, or perhaps even worse, run successfully but provide an inappropriate result.

Running external functions in WLM environments:

You can use the WLM ENVIRONMENT clause to identify the address space in which a function or is to run. Using different WLM environments lets you isolate one group of programs from another. For example, you might choose to isolate programs based on security requirements and place all payroll applications in one WLM environment because those applications deal with data, such as employee salaries.

To prevent a user from defining functions in sensitive WLM environments, DB2 invokes the external security manager to determine whether the user has authorization to issue CREATE FUNCTION statements that refer to the specified WLM environment. The following example shows the RACF command that authorizes DB2 user DB2USER1 to register a function on DB2 subsystem DB2A that runs in the WLM environment named PAYROLL.

```
PERMIT DB2A.WLMENV.PAYROLL CLASS(DSNR) ID(DB2USER1)  ACCESS(READ)
```

Scrollable cursors specified with user-defined functions:

A row can be fetched more than once with a scrollable cursor. Therefore, if a scrollable cursor is defined with a function that is not deterministic in the select list of the cursor, a row can be fetched multiple times with different results for each fetch. Similarly, if a scrollable cursor is defined with a user-defined function with external action, the action is executed with every fetch.

Creating a secure function:

Typically, the security administrator will examine the data that is accessed by a function, ensure that it is secure, and grant the CREATE_SECURE_OBJECT privilege to someone who currently requires the privileges to create a secure user-defined function. After the function is created, they will revoke the CREATE_SECURE_OBJECT privilege from the function owner.

Invoking other user-defined functions in a secure function:
If a secure user-defined function invokes other user-defined functions, DB2 does not validate whether those nested user-defined functions have the SECURED attribute. If those nested functions can access sensitive data, the security administrator needs to ensure that those functions are allowed to access the sensitive data and should ensure that a change control audit procedure has been established for all changes to those functions.

SECURE column in the DSN_FUNCTION_TABLE EXPLAIN table:
If a row permission or a column mask definition references a user-defined function, the user-defined function must be secure because the sensitive data might be passed as arguments to the function. The column SECURE in the EXPLAIN table DSN_FUNCTION_TABLE indicates whether a user-defined function is considered secure.

Functions and global variables:
The content of global variables that are referenced in functions is inherited from the caller. Global variables cannot be modified in or by functions.

Alternative syntax and synonyms:
To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords:
- VARIANT as a synonym for NOT DETERMINISTIC
- NOT VARIANT as a synonym for DETERMINISTIC
- NOT NULL CALL as a synonym for RETURNS NULL ON NULL INPUT
- NULL CALL as a synonym for CALLED ON NULL INPUT
- PARAMETER STYLE DB2SQL as a synonym for PARAMETER STYLE SQL
- TIMEZONE can be specified as an alternative to TIME ZONE.

Examples

Example 1: Assume that you want to write an external function program in C that implements the following logic:

```
output = 2 * input - 4
```

The function should return a null value if and only if one of the input arguments is null. The simplest way to avoid a function call and get a null result when an input value is null is to specify RETURNS NULL ON NULL INPUT on the CREATE FUNCTION statement or allow it to be the default. Write the statement needed to register the function, using the specific name MINENULL1.

```
CREATE FUNCTION NTEST1 (SMALLINT)
   RETURNS SMALLINT
   EXTERNAL NAME 'NTESTMOD'
   SPECIFIC MINENULL1
   LANGUAGE C
   DETERMINISTIC
   NO SQL
   FENCED
   PARAMETER STYLE SQL
   RETURNS NULL ON NULL INPUT
   NO EXTERNAL ACTION;
```

Example 2: Assume that user Smith wants to register an external function named CENTER in schema SMITH. The function program will be written in C and will be reentrant. Write the statement that Smith needs to register the function, letting DB2 generate a specific name for the function.

```
CREATE FUNCTION CENTER (INTEGER, FLOAT)
   RETURNS FLOAT
   EXTERNAL NAME 'MIDDLE'
   LANGUAGE C
   DETERMINISTIC
   NO SQL
   FENCED
   PARAMETER STYLE SQL
   NO EXTERNAL ACTION
   STAY RESIDENT YES;
```

Example 3: Assume that user McBride (who has administrative authority) wants to register an external function named CENTER in the SMITH schema. McBride plans to give the function specific name FOCUS98. The function program uses a scratchpad to perform some one-time only initialization and save the results. The function program returns a value with a FLOAT data type. Write the statement McBride needs to register the function and ensure that when the function is invoked, it returns a value with a data type of DECIMAL(8,4).

```
CREATE FUNCTION SMITH.CENTER (FLOAT, FLOAT, FLOAT)
   RETURNS DECIMAL(8,4) CAST FROM FLOAT
   EXTERNAL NAME 'CMOD'
   SPECIFIC FOCUS98
   LANGUAGE C
   DETERMINISTIC
   NO SQL
   FENCED
   PARAMETER STYLE SQL
   NO EXTERNAL ACTION
   SCRATCHPAD
   NO FINAL CALL;
```

Example 4: The following example registers a Java user-defined function that returns the position of the first vowel in a string. The user-defined function is written in Java, is to be run fenced, and is the FINDVWL method of class JAVAUDFS.

```
CREATE FUNCTION FINDV (CLOB(100K))
   RETURNS INTEGER
   FENCED
   LANGUAGE JAVA
   PARAMETER STYLE JAVA
   EXTERNAL NAME 'JAVAUDFS.FINDVWL'
   NO EXTERNAL ACTION
   CALLED ON NULL INPUT
   DETERMINISTIC
   NO SQL;
```

CREATE FUNCTION (external table)

This CREATE FUNCTION statement registers a user-defined external table function with a database server. A user-defined external table function can be used in the FROM clause of a subselect. It returns a table to the subselect by returning one row at a time each time it is invoked.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set defined below must include at least one of the following:

- The CREATEIN privilege on the schema
- SYSADM or SYSCTRL authority
- System DBADM

The authorization ID that matches the schema name implicitly has the CREATEIN privilege on the schema.

If the authorization ID that is used to create the function has installation SYSADM authority, the function is identified as system-defined function.

Additional privileges are required if the function uses a table as a parameter, refers to a distinct type, or is to run in a WLM (workload manager) environment. These privileges are:

- The SELECT privilege on any table that is an input parameter to the function.
- The USAGE privilege on each distinct type that the function references.
- Authority to create programs in the specified WLM environment. This authorization is obtained from an external security product, such as RACF.

At least one of the following additional privileges is required if the SECURED option is specified

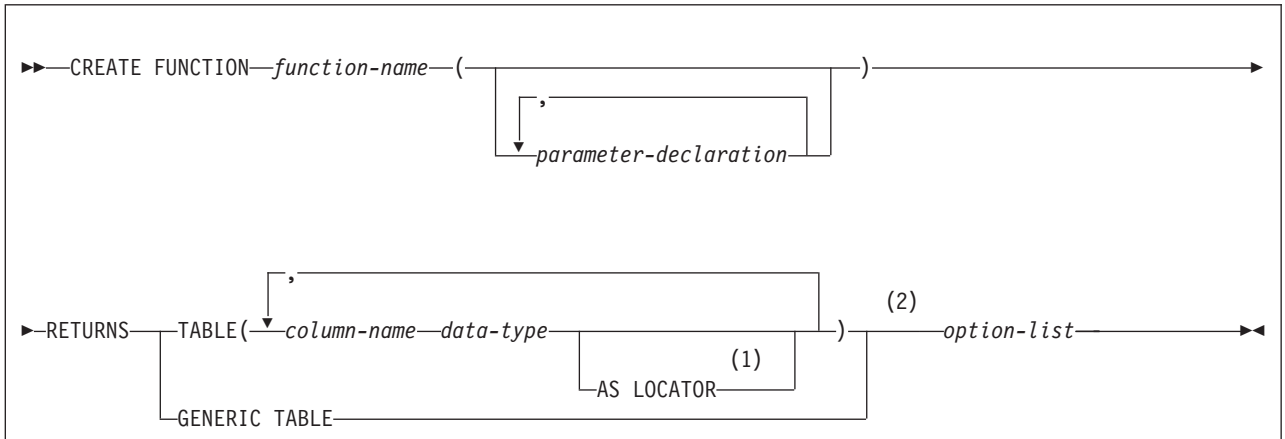
- SECADM authority
- CREATE_SECURE_OBJECT privilege

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the owner is a role, the implicit schema match does not apply and this role needs to include one of the previously listed conditions.

If the statement is dynamically prepared and is not running in a trusted context for which the ROLE AS OBJECT OWNER clause is specified, the privilege set is the set of privileges that are held by the SQL authorization ID of the process. If the schema name is not the same as the SQL authorization ID of the process, one of the following conditions must be met:

- The privilege set includes SYSADM or SYSCTRL authority.
- The SQL authorization ID of the process has the CREATEIN privilege on the schema.

Syntax

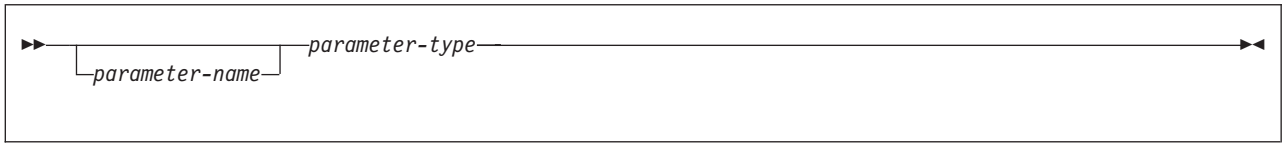


Notes:

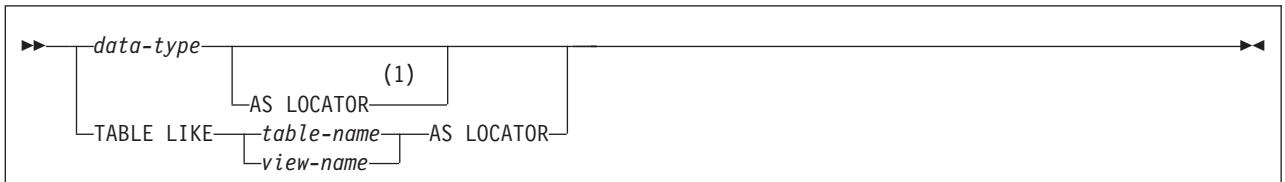
1 AS LOCATOR can be specified only for a LOB data type or a distinct type based on a LOB data type.

2 This clause and the clauses that follow in the *option-list* can be specified in any order.

parameter-declaration:



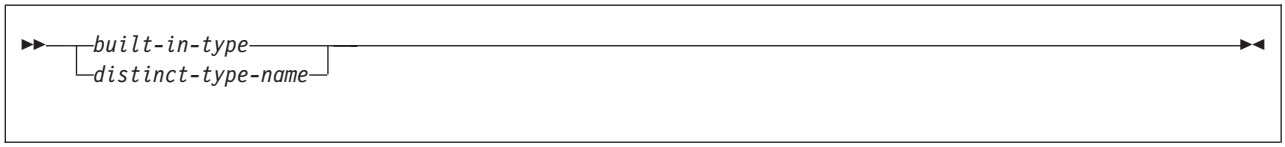
parameter-type:



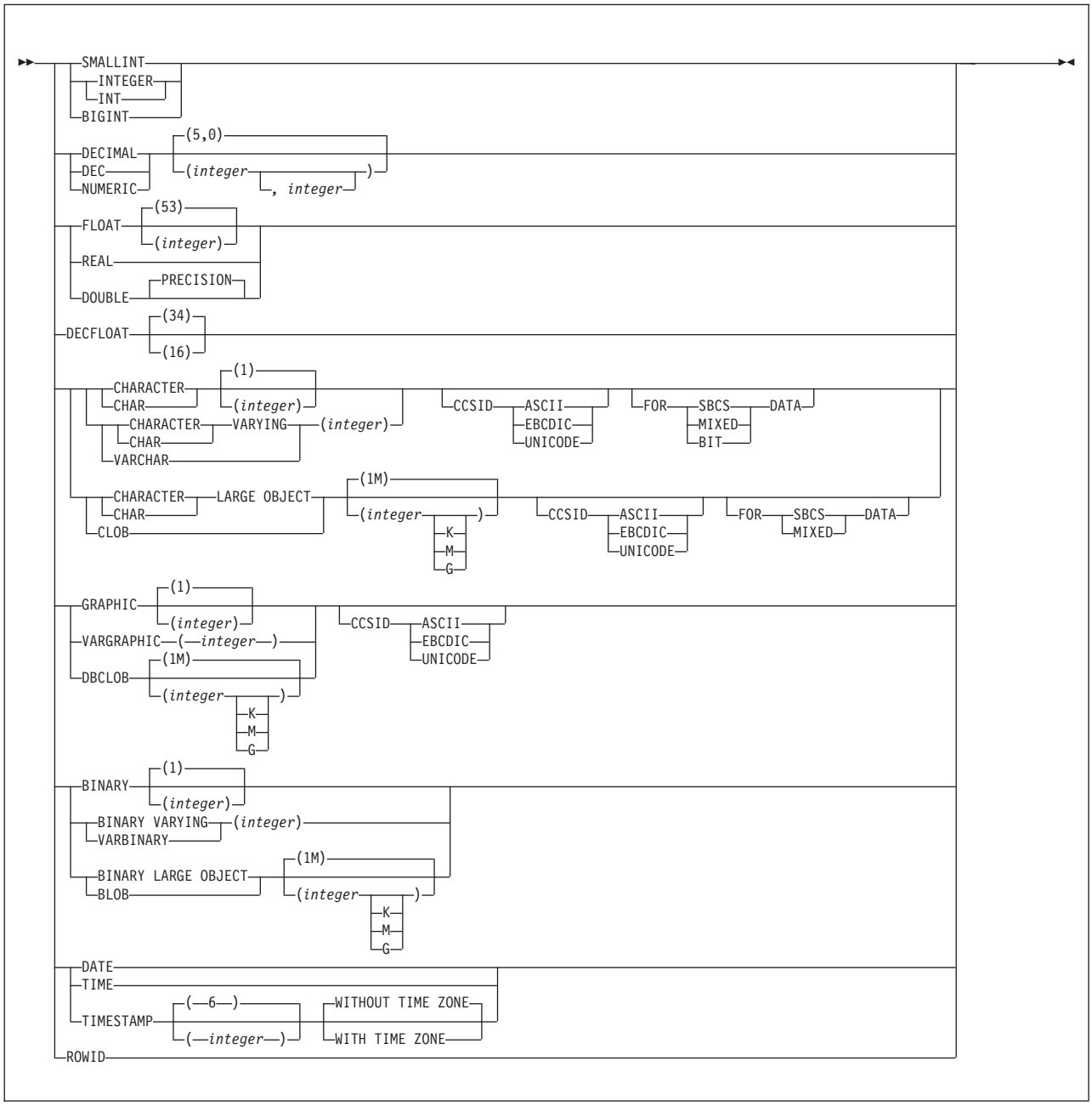
Notes:

1 AS LOCATOR can be specified only for a LOB data type or a distinct type based on a LOB data type.

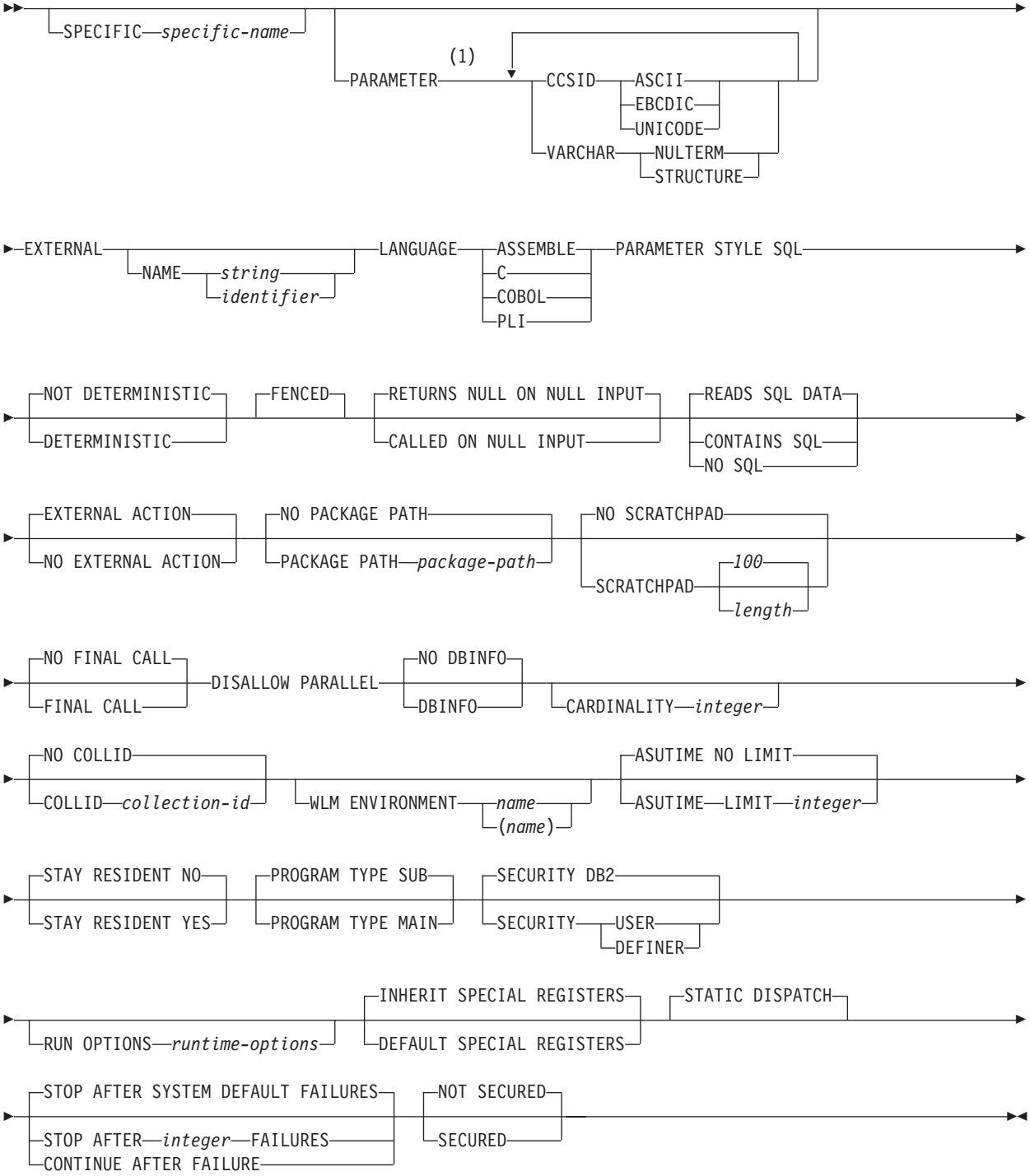
data-type:



built-in-type:



option-list: (The clauses in the *option-list* can be specified in any order.)



Notes:

1 The same clause must not be specified more than one time.

Description

function-name
Names the user-defined function. The name is implicitly or explicitly qualified by a schema name.

The combination of name, schema name, the number of parameters, and the data type of each parameter[27] (without regard for any length, precision, scale, subtype or encoding scheme attributes of the data type) must not identify a user-defined function that exists at the current server.

You can use the same name for more than one function if the function signature of each function is unique.

- The unqualified form of *function-name* must not be any of the following system-reserved keywords even if you specify them as delimited identifiers:

```
ALL                 LIKE                UNIQUE
AND                 MATCH               UNKNOWN
ANY                 NOT                 =
BETWEEN             NULL                ¬=
DISTINCT            ONLY                <
EXCEPT              OR                  <=
EXISTS              OVERLAPS            ¬<
FALSE               SIMILAR             >
FOR                 SOME                >=
FROM                TABLE               ¬>
IN                  TRUE                <>
IS                  TYPE
```

The schema name can be 'SYSTOOLS' or 'SYSFUN' if the user who executes the CREATE statement has SYSADM or SYSCTRL privilege. Otherwise, the schema name must not begin with 'SYS' unless the schema name is 'SYSADM'.

(*parameter-declaration***,...)**

Identifies the number of input parameters of the function, and specifies the data type of each parameter. All of the parameters for a function are input parameters and are nullable. There must be one entry in the list for each parameter that the function expects to receive. Although not required, you can give each parameter a name.

A function can have no parameters. In this case, you must code an empty set of parentheses, for example:

```
CREATE FUNCTION WOOFER()
```

parameter-name

Specifies the name of the input parameter. The name is an SQL identifier, and each name in the parameter list must not be the same as any other name. The same name cannot be used for a parameter name and a column name.

data-type

Specifies the data type of the input parameter. The data type can be a built-in data type or a user-defined type.

built-in-type

The data type of the input parameter is a built-in data type.

For information on the data types, see built-in-type.

For parameters with a character or graphic data type, the PARAMETER CCSID clause or CCSID clause indicates the encoding scheme of the parameter. If you do not specify either of these clauses, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

distinct-type-name

The data type of the input parameter is a distinct type. Any length,

27. If the function has more than 30 parameters, only the first 30 parameters are used to determine whether the function is unique.

precision, scale, subtype, or encoding scheme attributes for the parameter are those of the source type of the distinct type.

If you specify the name of the distinct type without a schema name, DB2 resolves the schema name by searching the schemas in the SQL path.

Although parameters with a character data type have an implicitly or explicitly specified subtype (BIT, SBCS, or MIXED), the function program can receive character data of any subtype. Therefore, conversion of the input data to the subtype of the parameter might occur when the function is invoked. An error occurs if mixed data that actually contains DBCS characters is used as the value for an input parameter that is declared with an SBCS subtype.

Parameters with a datetime data type or a distinct type are passed to the function as a different data type:

- A datetime type parameter is passed as a character data type, and the data is passed in ISO format.

  The encoding scheme for a datetime type parameter is the same as the implicitly or explicitly specified encoding scheme of any character or graphic string parameters. If no character or graphic string parameters are passed, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.
- A distinct type parameter is passed as the source type of the distinct type.

AS LOCATOR
Specifies that a locator to the value of the parameter is passed to the function instead of the actual value. Specify AS LOCATOR only for parameters with a LOB data type or a distinct type that is based on a LOB data type. Passing locators instead of values can result in fewer bytes being passed to the function, especially when the value of the parameter is very large.

The AS LOCATOR clause has no effect on determining whether data types can be promoted, nor does it affect the function signature, which is used in function resolution.

TABLE LIKE *table-name* **or** *view-name* **AS LOCATOR**
Specifies that the parameter is a transition table. However, when the function is invoked, the actual values in the transition table are not passed to the function. A single value is passed instead. This single value is a locator to the table, which the function uses to access the columns of the transition table. A function with a table parameter can only be invoked from the triggered action of a trigger.

The use of TABLE LIKE provides an implicit definition of the transition table. It specifies that the transition table has the same number of columns as the identified table or view. If a table is specified, the transition table includes columns that are defined as implicitly hidden in the table. The columns have the same data type, length, precision, scale, subtype, and encoding scheme as the identified table or view, as they are described in catalog tables SYSCOLUMNS and SYSTABLESPACE. The number of columns and the attributes of those columns are determined at the time the CREATE FUNCTION statement is processed. Any subsequent changes to the number of columns in the table or the attributes of those columns do not affect the parameters of the function.

table-name or *view-name* must identify a table or view that exists at the current server. A view cannot have columns of length 0. The name must not identify a declared temporary table. The table that is identified can contain XML columns; however, the function cannot reference those XML columns. The name does not have to be the same name as the table that is associated with the transition table for the trigger. An unqualified table or view name is implicitly qualified according to the following rules:

- If the CREATE FUNCTION statement is embedded in a program, the implicit qualifier is the authorization ID in the QUALIFIER bind option when the plan or package was created or last rebound. If QUALIFIER was not used, the implicit qualifier is the owner of the plan or package.
- If the CREATE FUNCTION statement is dynamically prepared, the implicit qualifier is the SQL authorization ID in the CURRENT SCHEMA special register.

When the function is invoked, the corresponding columns of the transition table identified by the table locator and the table or view identified in the TABLE LIKE clause must have the same definition. The data type, length, precision, scale, and encoding scheme of these columns must match exactly. The description of the table or view at the time the CREATE FUNCTION statement was executed is used.

Additionally, a character FOR BIT DATA column of the transition table cannot be passed as input for a table parameter for which the corresponding column of the table specified at the definition is not defined as character FOR BIT DATA. (The definition occurs with the CREATE FUNCTION statement.) Likewise, a character column of the transition table that is not FOR BIT DATA cannot be passed as input for a table parameter for which the corresponding column of the table specified at the definition is defined as character FOR BIT DATA.

For more information about using table locators, see *DB2 Application Programming and SQL Guide*.

RETURNS TABLE(*column-name data-type* **...)**

Identifies that the output of the function is a table. The parentheses that follow the keyword enclose the list of names and data types of the columns of the table.

column-name

Specifies the name of the column. The name is an SQL identifier and must be unique within the RETURNS TABLE clause for the function.

data-type

Specifies the data type of the column. The column is nullable.

AS LOCATOR

Specifies that the function returns a locator to the value rather than the actual value. You can specify AS LOCATOR only for a LOB data type or a distinct type based on a LOB data type.

RETURNS GENERIC TABLE

Specifies that the output of the function is a generic table. This option can only be specified if LANGUAGE C is also specified.

The names and data types of the columns must be declared when the table function is references using the *typed-correlation-clause* of the subselect.

Related information:

"typed-correlation-clause" on page 786

SPECIFIC *specific-name*

Specifies a unique name for the function. The name is implicitly or explicitly qualified with a schema name. The name, including the schema name, must not identify the specific name of another function that exists at the current server.

The unqualified form of *specific-name* is an SQL identifier. The qualified form is an SQL identifier (the schema name) followed by a period and an SQL identifier.

If you do not specify a schema name, it is the same as the explicit or implicit schema name of the function name (*function-name*). If you specify a schema name, it must be the same as the explicit or implicit schema name of the function name.

If you do not specify the SPECIFIC clause, the default specific name is the name of the function. However, if the function name does not provide a unique specific name or if the function name is a single asterisk, DB2 generates a specific name in the form of:

```
SQLxxxxxxxxxxxx
```

where `'xxxxxxxxxxxx'` is a string of 12 characters that make the name unique.

The specific name is stored in the SPECIFIC column of the SYSROUTINES catalog table. The specific name can be used to uniquely identify the function in several SQL statements (such as ALTER FUNCTION, COMMENT, DROP, GRANT, and REVOKE) and in DB2 commands (START FUNCTION, STOP FUNCTION, and DISPLAY FUNCTION). However, the function cannot be invoked by its specific name.

PARAMETER CCSID or VARCHAR

CCSID

Indicates whether the encoding scheme for character or graphic string parameters is ASCII, EBCDIC, or UNICODE. The default encoding scheme is the value specified in the CCSID clauses of the parameter list or RETURNS TABLE clause, or in the field DEF ENCODING SCHEME on installation panel DSNTIPF.

This clause provides a convenient way to specify the encoding scheme for character or graphic string parameters. If individual CCSID clauses are specified for individual parameters in addition to this PARAMETER CCSID clause, the value specified in *all* of the CCSID clauses must be the same value that is specified in this clause.

This clause also specifies the encoding scheme to be used for system-generated parameters of the routine such as message tokens and DBINFO.

VARCHAR

Specifies that the representation of the values of varying length character string-parameters, including, if applicable, the output of the function, for functions which specify LANGUAGE C.

This option can only be specified if LANGUAGE C is also specified.

NULTERM

Specifies that variable length character string parameters are represented in a NUL-terminated string form.

STRUCTURE
Specifies that variable length character string parameters are represented in a VARCHAR structure form.

Using the PARAMETER VARCHAR clause, there is no way to specify the VARCHAR form of an individual parameter as these is with PARAMETER CCSID. The PARAMETER VARCHAR clause only applies to parameters in the parameter list of a function and in the RETURNS TABLE clause. It does not apply to system-generated parameters of the routine such as message tokens and DBINFO.

In a data sharing environment, you should not specify the PARAMETER VARCHAR clause until all members of the data sharing group support the clause. If some group members support this clause and others do not, and PARAMETER VARCHAR is specified in an external routine, the routine will encounter different parameter forms depending on which group member invokes the routine.

EXTERNAL
Specifies that the function being registered is based on code that is written in an external programming language and adheres to the documented linkage conventions and interface of that language.

If you do not specify the NAME clause, 'NAME *function-name*' is implicit. In this case, *function-name* must not be longer than 8 characters.

NAME *string* **or** *identifier*
Identifies the name of the load module that contains the user-written code that implements the logic of the function.

For other values of LANGUAGE, the name can be a string constant that is no longer than 8 characters. It must conform to the naming conventions for load modules. Alphabetical extenders for national languages can be used as the first character and as subsequent characters in the load module name.

DB2 loads the load module when the function is invoked. The load module is created when the program that contains the function body is compiled and link-edited. The load module does not need to exist when the CREATE FUNCTION statement is executed. However, it must exist and be accessible by the current server when the function is invoked.

You can specify the EXTERNAL clause in one of the following ways:

```
EXTERNAL
EXTERNAL NAME PKJVSP1
EXTERNAL NAME 'PKJVSP1'
```

If you specify an external program name, you must use the NAME keyword. For example, this syntax is not valid:

```
EXTERNAL PKJVSP1
```

LANGUAGE
Specifies the application programming language in which the function program is written. All programs must be designed to run in IBM's Language Environment environment.

ASSEMBLE
The function is written in Assembler.

C The function is written in C or C++. The VARCHAR clause can only be specified is LANGUAGE C is specified.

COBOL
The function is written in COBOL, including the object-oriented language extensions.

PLI
The function is written in PL/I.

PARAMETER STYLE SQL
Specifies the linkage convention that the function program uses to receive input parameters from and pass return values to the invoking SQL statement.

PARAMETER STYLE SQL specifies the parameter passing convention that supports passing null values both as input and for output.

If the RETURNS TABLE clause is specified, the parameters that are passed between the invoking SQL statement and the function include:

- *n* parameters for the input parameters that are specified for the function
- *m* parameters for the result columns of the function that are specified on the RETURNS TABLE clause
- *n* parameters for the indicator variables for the input parameters
- *m* parameters for the indicator variables of the result columns of the function that are specified on the RETURNS TABLE clause
- The SQLSTATE to be returned to DB2
- The qualified name of the function
- The specific name of the function
- The SQL diagnostic string to be returned to DB2
- The scratchpad, if SCRATCHPAD is specified
- The call type
- The DBINFO structure, if DBINFO is specified

If the RETURNS GENERIC TABLE clause is specified, the parameters that are passed between the invoking SQL statement and the function include:

- *n* parameters for the input parameters that are specified for the function
- *n* parameters for the indicator variables for the input parameters
- *m* parameters for the result columns of the function that are specified on the RETURNS GENERIC TABLE clause
- A result table descriptor that contains the following:
  - *m* result columns of the function that are specified in the *typed-correlation-clause* of the *table-function-reference* in a SELECT statement
  - An array of *m*, 4-byte addresses to the values of the result columns
  - An array of *m*, null indicators of the result columns
- The SQLSTATE to be returned to DB2
- The qualified name of the function
- The specific name of the function
- The SQL diagnostic string to be returned to DB2
- The scratchpad, if SCRATCHPAD is specified
- The call type
- The DBINFO structure, if DBINFO is specified

For complete details about the structure of the parameter list that is passed, see *DB2 Application Programming and SQL Guide*.

NOT DETERMINISTIC or DETERMINISTIC

Specifies whether the function returns function returns the same results each time that the function is invoked with the same input arguments.

NOT DETERMINISTIC

The function might not return the same results each time that the function is invoked with the same input arguments. The function depends on some state values that affect the results. DB2 uses this information to disable the merging of views and table expressions when processing SELECT and SQL data change statements that refer to this function. An example of a function that is not deterministic is one that generates random numbers, or any function that contains SQL statements.

NOT DETERMINISTIC is the default.

DETERMINISTIC

The function always returns the same result each time that the function is invoked with the same input arguments. An example of a deterministic function is a function that calculates the square root of the input. DB2 uses this information to enable the merging of views and table expressions for SELECT and SQL data change statements that refer to this function. DETERMINISTIC is not the default. If applicable, specify DETERMINISTIC to prevent non-optimal access paths from being chosen for SQL statements that refer to this function.

DB2 does not verify that the function program is consistent with the specification of DETERMINISTIC or NOT DETERMINISTIC.

FENCED

Specifies that the function runs in an external address space to prevent the function from corrupting DB2 storage.

FENCED is the default.

RETURNS NULL ON NULL INPUT or CALLED ON NULL INPUT

Specifies whether the function is called if any of the input arguments is null at execution time.

RETURNS NULL ON NULL INPUT

The function is not called if any of the input arguments is null. The result is an empty table, which is a table with no rows. RETURNS NULL ON INPUT is the default.

CALLED ON NULL INPUT

The function is called regardless of whether any of the input arguments are null, making the function responsible for testing for null argument values. The function can return an empty table, depending on its logic.

READS SQL DATA, CONTAINS SQL, or NO SQL

Specifies which SQL statements, if any, can be executed in the function or any routine that is called from this function. The default is READS SQL DATA. For the data access classification of each statement, see Table 162 on page 2030.

READS SQL DATA

Specifies that the function can execute statements with a data access indication of READS SQL DATA, CONTAINS SQL, or NO SQL. The function cannot execute SQL statements that modify data.

CONTAINS SQL
Specifies that the function can execute only SQL statements with an access indication of CONTAINS SQL or NO SQL. The function cannot execute statements that read or modify data.

NO SQL
Specifies that the function can execute only SQL statements with a data access classification of NO SQL.

EXTERNAL ACTION or NO EXTERNAL ACTION
Specifies whether the function takes an action that changes the state of an object that DB2 does not manage. An example of an external action is sending a message or writing a record to a file.

Because DB2 uses the RRS attachment for functions, DB2 can participate in two-phase commit with any other resource manager that uses RRS. For resource managers that do not use RRS, there is no coordination of commit or rollback operations on non-DB2 resources.

EXTERNAL ACTION
The function can take an action that changes the state of an object that DB2 does not manage.

If you specify EXTERNAL ACTION, DB2:

- Materializes the views and table expressions in SELECT and SQL data change statements that refer to the function. This materialization can adversely affect the access paths that are chosen for the SQL statements that refer to this function. Do not specify EXTERNAL ACTION if the function does not have an external action.
- Does not move the function from one task control block (TCB) to another between FETCH operations.
- Does not allow another function or stored procedure to use the TCB until the cursor is closed. This is also applicable for cursors declared WITH HOLD.

The only changes to resources made outside of DB2 that are under the control of commit and rollback operations are those changes made under RRS control.

EXTERNAL ACTION is the default.

NO EXTERNAL ACTION
The function does not take any action that changes the state of an object that DB2 does not manage. DB2 uses this information to enable the merging of views and table expressions for SELECT and SQL data change statements that refer to this function. If applicable, specify NO EXTERNAL ACTION to prevent non-optimal access paths from being chosen for SQL statements that refer to this function.

Although the scope of global variables are beyond the scope of the routine, global variables can be set in the routine body when NO EXTERNAL ACTION is specified.

NO PACKAGE PATH or PACKAGE PATH *package-path*
Specifies the packagecpath to use when the function is run. This is the list of the possible package collections into which the DBRM this is associated with the function is bound.

NO PACKAGE PATH
Specifies that the list of package collections for the function is the same as the list of package collection IDs for the program that invokes the function.

If the program that invokes the function does not use a package, DB2 resolves the package by using the CURRENT PACKAGE PATH special register, the CURRENT PACKAGESET special register, or the PKLIST bind option (in this order). For information about how DB2 uses these three items, see *DB2 Application Programming and SQL Guide*.

PACKAGE PATH *package-path*
Specifies a list of package collections, in the same format as the SET CURRENT PACKAGE PATH special register.

If the COLLID clause is specified with PACKAGE PATH, the COLLID clause is ignored when the function is invoked.

The *package-path* value that is provided when the function is created is checked when the function is invoked. If *package-path* contains SESSION_USER (or USER), PATH, or PACKAGE PATH, an error is returned when the *package-path* value is checked.

NO SCRATCHPAD or SCRATCHPAD
Specifies whether DB2 provides a scratchpad for the function. It is strongly recommended that functions be reentrant, and a scratchpad provides an area for the function to save information from one invocation to the next.

NO SCRATCHPAD
Specifies that a scratchpad is not allocated and passed to the function. NO SCRATCHPAD is the default.

SCRATCHPAD *length*
Specifies that when the function is invoked for the first time, DB2 allocates memory for a scratchpad. A scratchpad has the following characteristics:

- *length* must be between 1 and 32767. The default value is 100 bytes.
- DB2 initializes the scratchpad to all binary zeros (X'00"s).
- The scope of a scratchpad is the SQL statement. Each reference to the function in an SQL statement has a scratchpad. For example, assuming that function UDFX was defined with the SCRATCHPAD keyword, two scratchpads are allocated for the two references to UDFX in the following SQL statement:

```
SELECT *
FROM TABLE (UDFX(A)), TABLE (UDFX(B));
```

- The scratchpad is persistent. DB2 preserves its content from one invocation of the function to the next. Any changes that the function makes to the scratchpad on one call are still there on the next call. DB2 initializes the scratchpads when it begins to execute an SQL statement. DB2 does not reset scratchpads when a correlated subquery begins to execute.
- The scratchpad can be a central point for the system resources that the function acquires. If the function acquires system resources, specify FINAL CALL to ensure that DB2 calls the function one more time so that the function can free those system resources.

Each time the function invoked, DB2 passes an additional argument to the function that contains the address of the scratchpad.

If you specify SCRATCHPAD, DB2:

- Does not move the function from one task control block (TCB) to another between FETCH operations.

- Does not allow another function or stored procedure to use the TCB until the cursor is closed. This is also applicable for cursors declared WITH HOLD.

NO FINAL CALL or FINAL CALL
Specifies whether a *first call* and a *final call* are made to the function.

NO FINAL CALL
A first call and final call are not made to the function. NO FINAL CALL is the default.

FINAL CALL
A first call and final call are made to the function in addition to one or more *open*, *fetch*, or *close calls.*

The types of calls are:

First call
A *first call* occurs only if the function was defined with FINAL CALL. Before a first call, the scratchpad is set to binary zeros. Argument values are passed to the function, and the function might acquire memory or perform other one-time only resource initialization. However, the function should not return any data to DB2, but it can set return values for the SQL-state and diagnostic-message arguments.

Open call
An *open call* occurs unless the function returns an error. The scratchpad is set to binary zeros only if the function was defined with NO FINAL CALL. Argument values are passed to the function, and the function might perform any one-time initialization actions that are required. However, the function should not return any data to DB2.

Fetch call
A *fetch call* occurs unless the function returns an error during the first call or open call. Argument values are passed to the function, and DB2 expects the function to return a row of data or the end-of-table condition. If a scratchpad is also passed to the function, it remains untouched from the previous call.

Close call
A *close call* occurs unless the function returns an error during the first call, open call, or fetch call. No SQL-argument or SQL-argument-ind values are passed to the function, and if the function attempts to examine these values, unpredictable results might occur. If a scratchpad is also passed to the function, it remains untouched from the previous call.

The function should not return any data to DB2, but it can set return values for the SQL-state and diagnostic-message arguments. Also on close call, a function that is defined with NO FINAL CALL should release any system resources that it acquired. (A function that is defined with FINAL CALL should release any acquired resources on the final call.)

Final The *final call* balances the first call, and like the first call, occurs only if the function was defined with FINAL CALL. The function can set return values for the SQL-state and diagnostic-message arguments. The function should also release any system resources that it acquired. A final call occurs at these times:

- *End of statement:* When the cursor is closed for cursor-oriented statements, or the execution of the statement has completed.
- *End of transaction:* When normal end of statement processing does not occur. For example, the logic of an application, for some reason, bypasses closing the cursor.

If a commit, rollback, or abort operation causes the final call, the function cannot issue any SQL statements when it is invoked.

DISALLOW PARALLEL
Specifies that DB2 does not consider parallelism for the function.

NO DBINFO or DBINFO
Specifies whether additional status information is passed to the function when it is invoked.

NO DBINFO
No additional information is passed. NO DBINFO is the default.

DBINFO
An additional argument is passed when the function is invoked. The argument is a structure that contains information such as the application run time authorization ID, the schema name, the name of a table or column that the function might be inserting into or updating, and identification of the database server that invoked the function. For details about the argument and its structure, see *DB2 Application Programming and SQL Guide*.

CARDINALITY *integer*
Specifies an estimate of the expected number of rows that the function returns. The number is used for optimization purposes. The value of *integer* must range from 0 to 2147483647.

If you do not specify CARDINALITY, DB2 assumes a finite value. The finite value is the same value that DB2 assumes for tables for which the RUNSTATS utility has not gathered statistics.

If a function has an infinite cardinality—the function never returns the "end-of-table" condition and always returns a row, then a query that requires the "end-of-table" to work correctly will need to be interrupted. Thus, avoid using such functions in queries that involve GROUP BY and ORDER BY.

NO COLLID or COLLID *collection-id*
Identifies the package collection that is to be used when the function is executed. This is the package collection into which the DBRM that is associated with the function program is bound.

NO COLLID
The package collection for the function is the same as the package collection of the program that invokes the function. If a trigger invokes the function, the collection of the trigger package is used. If the invoking program does not use a package, DB2 resolves the package by using the CURRENT PACKAGE PATH special register, the CURRENT PACKAGESET special register, or the PKLIST bind option (in this order). For details about how DB2 uses these three items, see the information on package resolution in *DB2 Application Programming and SQL Guide*.

NO COLLID is the default.

COLLID *collection-id*
The name of the package collection that is to be used when the external is executed.

WLM ENVIRONMENT

Identifies the WLM (workload manager) application environment in which the function is to run. The *name* of the WLM environment is an SQL identifier.

If you do not specify WLM ENVIRONMENT, the function runs in the WLM-established stored procedure address space that is specified at installation time.

name

The WLM environment in which the function must run. If another user-defined function or a stored procedure calls the function and that calling routine is running in an address space that is not associated with the WLM environment, DB2 routes the function request to a different address space.

(name,*)

When an SQL application program directly invokes the function, the WLM environment in which the function runs.

If another user-defined function or a stored procedure calls the function, the function runs in same environment that the calling routine uses. In this case, authorization to run the function in the WLM environment is not checked because the authorization of the calling routine suffices.

Users must have the appropriate authorization to execute functions in the specified WLM environment. For an example of a RACF command that provides this authorization, see Running external functions in WLM environments.

ASUTIME

Specifies the total amount of processor time, in CPU service units, that a single invocation of the function can run. The value is unrelated to the ASUTIME column of the resource limit specification table.

When you are debugging a function, setting a limit can be helpful if the function gets caught in a loop. For information on service units, see *z/OS MVS Initialization and Tuning Guide*.

NO LIMIT

There is no limit on the service units. NO LIMIT is the default.

LIMIT *integer*

The limit on the number of CPU service units is a positive *integer* in the range of 1 to 2 147 483 647. If the procedure uses more service units than the specified value, DB2 cancels the procedure. The CPU cycles that are consumed by parallel tasks in a procedure do not contribute towards the specified ASUTIME LIMIT.

STAY RESIDENT

Specifies whether the load module for the function is to remain resident in memory when the function ends.

NO The load module is deleted from memory after the function ends. Use NO for non-reentrant functions. NO is the default.

YES

The load module remains resident in memory after the function ends. Use YES for reentrant functions.

PROGRAM TYPE

Specifies whether the function program runs as a main routine or a subroutine.

SUB
The function runs as a subroutine. SUB is the default.

MAIN
The function runs as a main routine.

SECURITY
Specifies how the function interacts with an external security product, such as RACF, to control access to non-SQL resources.

DB2
The function does not require an external security environment. If the function accesses resources that an external security product protects, the access is performed using the authorization ID that is associated with the WLM-established stored procedure address space.

DB2 is the default.

USER
An external security environment should be established for the function. If the function accesses resources that the external security product protects, the access is performed using the primary authorization ID of the process that invoked the function.

DEFINER
An external security environment should be established for the function. If the function accesses resources that the external security product protects, the access is performed using the authorization ID of the owner of the function.

RUN OPTIONS *runtime-options*
Specifies the Language Environment run time options to be used for the function. You must specify *runtime-options* as a character string that is no longer than 254 bytes. If you do not specify RUN OPTIONS or pass an empty string, DB2 does not pass any run time options to Language Environment, and Language Environment uses its installation defaults.

For a description of the Language Environment run time options, see *z/OS Language Environment Programming Reference*.

INHERIT SPECIAL REGISTERS or DEFAULT SPECIAL REGISTERS
Specifies how special registers are set on entry to the routine.

INHERIT SPECIAL REGISTERS
Specifies that the values of special registers are inherited according to the rules listed in the table for characteristics of special registers in a user-defined function in Table 40 on page 205.

DEFAULT SPECIAL REGISTERS
Specifies that special registers are initialized to the default values, as indicated by the rules in the table for characteristics of special registers in a user-defined function in Table 40 on page 205.

STATIC DISPATCH
At function resolution time, DB2 chooses a function based on the static (or declared) types of the function parameters. STATIC DISPATCH is the default.

STOP AFTER SYSTEM DEFAULT FAILURES, STOP AFTER *nn* **FAILURES, or CONTINUE AFTER FAILURE**
Specifies whether the routine is to be put in a stopped state after some number of failures.

STOP AFTER SYSTEM DEFAULT FAILURES
Specifies that this routine should be placed in a stopped state after the number of failures indicated by the value of field MAX ABEND COUNT on installation panel DSNTIPX. This is the default.

STOP AFTER *nn* **FAILURES**
Specifies that this routine should be placed in a stopped state after *nn* failures. The value *nn* can be an integer from 1 to 32767.

CONTINUE AFTER FAILURE
Specifies that this routine should not be placed in a stopped state after any failure.

NOT SECURED or SECURED
Specifies if the function is considered secure for row access control and column access control.

NOT SECURED
Specifies that the function is not considered as secure for row access control and column access control.

NOT SECURED is the default.

When the function is invoked, the arguments of the function must not reference a column for which a column mask is enabled when the table is using active column access control.

SECURED
Specifies that the function is considered secure for row access control and column access control.

The function must be defined with SECURED when it is referenced in a row permission or a column mask.

Notes

See "Notes" on page 1186 for information about:

- Owner privileges
- Choosing data types for parameters
- Specifying the encoding scheme for parameters
- Determining the uniqueness of functions in a schema
- Character string representation considerations
- Overriding a built-in function
- Scrollable cursors specified with user-defined functions
- Creating a secure function
- Invoking other user-defined functions in a secure function
- SECURE column in the DSN_FUNCTION_TABLE EXPLAIN table
- Functions and global variables

Determining if a table function is a generic table function:
To identify if a table function is a generic table function, you can query the SYSIBM.SYSROUTINES catalog table. The function is a generic table function if the value of the RESULT_COLS column is 0 (zero) when the value of the ROUTINETYPE column if 'F' and the value of the FUNCTIONTYPE column is 'T'.

Alternative syntax and synonyms:

To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords:

- VARIANT as a synonym for NOT DETERMINISTIC
- NOT VARIANT as a synonym for DETERMINISTIC
- NOT NULL CALL as a synonym for RETURNS NULL ON NULL INPUT
- NULL CALL as a synonym for CALLED ON NULL INPUT
- PARAMETER STYLE DB2SQL as a synonym for PARAMETER STYLE SQL
- TIMEZONE can be specified as an alternative to TIME ZONE.

Example

Example 1: The following example registers a table function written to return a row consisting of a single document identifier column for each known document in a text management system. The first parameter matches a given subject area and the second parameter contains a given string.

Within the context of a single session, the table function always returns the same table; therefore, it is defined as DETERMINISTIC. In addition, the DISALLOW PARALLEL keyword is added because table functions cannot operate in parallel.

Although the size of the output for DOCMATCH is highly variable, CARDINALITY 20 is a representative value and is specified to help DB2.

```
CREATE FUNCTION DOCMATCH (VARCHAR(30), VARCHAR(255))
                    RETURNS TABLE (DOC_ID CHAR(16))
  EXTERNAL NAME ABC
  LANGUAGE C
  PARAMETER STYLE SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION
  FENCED
  SCRATCHPAD
  FINAL CALL
  DISALLOW PARALLEL
  CARDINALITY 20;
```

Example 2: The following example registers a generic table function:

```
CREATE FUNCTION tf6(p1 VARCHAR(10))
 RETURNS GENERIC TABLE
 EXTERNAL NAME 'tf6'
 LANGUAGE C
 PARAMETER STYLE SQL
 DETERMINISTIC
 NO EXTERNAL ACTION
 FENCED
 SCRATCHPAD
 FINAL CALL;
```

Note that LANGUAGE C must be specified, and the names and data type of the result columns must be declared when the table function is referenced in the SELECT clause.

CREATE FUNCTION (sourced)

This CREATE FUNCTION statement registers a user-defined function that is based on an existing scalar or aggregate function with a database server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set defined below must include at least one of the following:

- The CREATEIN privilege on the schema
- SYSADM or SYSCTRL authority
- System DBADM

The authorization ID that matches the schema name implicitly has the CREATEIN privilege on the schema.

If the authorization ID that is used to create the function has installation SYSADM authority, the function is identified as system-defined function.

Additional privileges are required for the source function, and other privileges are also needed if the function uses a table as a parameter, or refers to a distinct type. These privileges are:

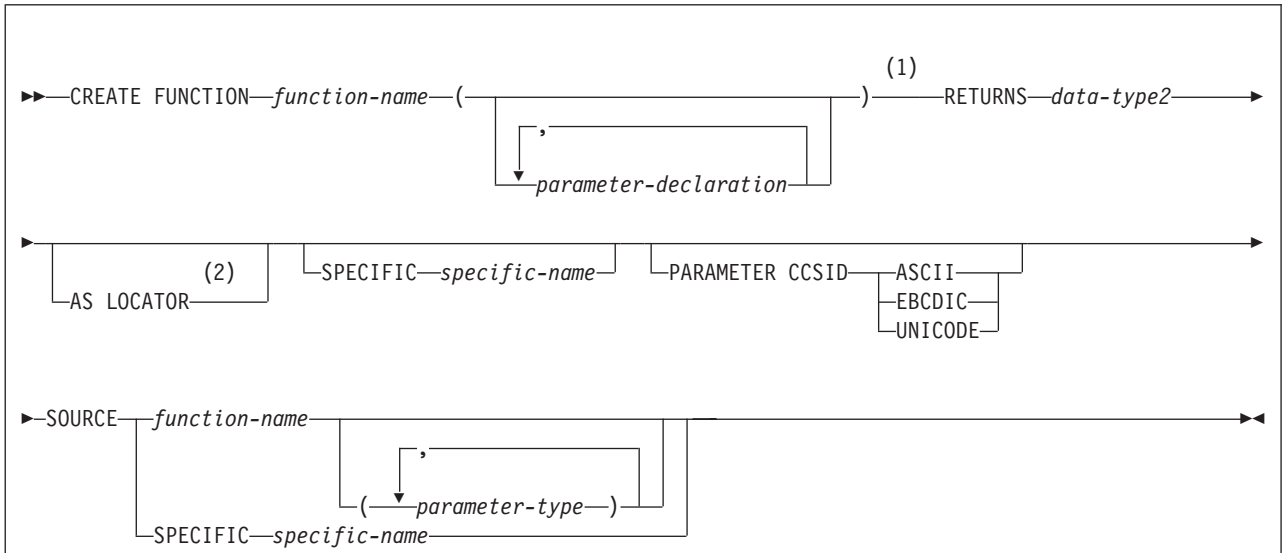
- The EXECUTE privilege for the function that the SOURCE clause references.
- The SELECT privilege on any table that is an input parameter to the function.
- The USAGE privilege on each distinct type that the function references.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the owner is a role, the implicit schema match does not apply and this role needs to include one of the previously listed conditions.

If the statement is dynamically prepared and is not running in a trusted context for which the ROLE AS OBJECT OWNER clause is specified, the privilege set is the set of privileges that are held by the SQL authorization ID of the process. If the schema name is not the same as the SQL authorization ID of the process, one of the following conditions must be met:

- The privilege set includes SYSADM or SYSCTRL authority.
- The SQL authorization ID of the process has the CREATEIN privilege on the schema.

Syntax

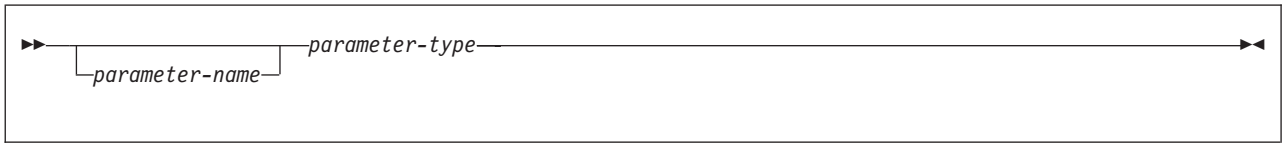


Notes:

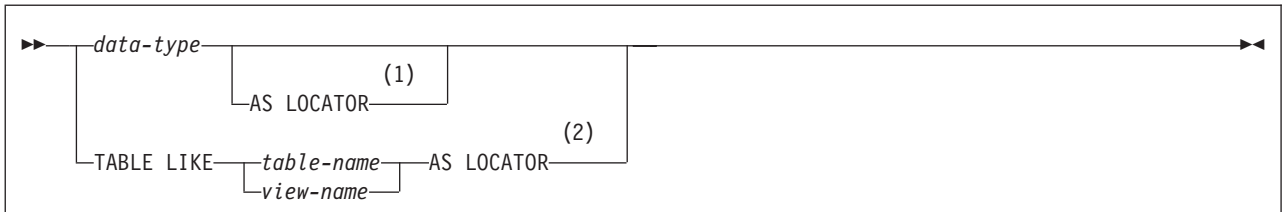
1 RETURNS, SPECIFIC, and SOURCE can be specified in any order.

2 AS LOCATOR can be specified only for a LOB data type or a distinct type based on a LOB data type.

parameter-declaration:



parameter-type:

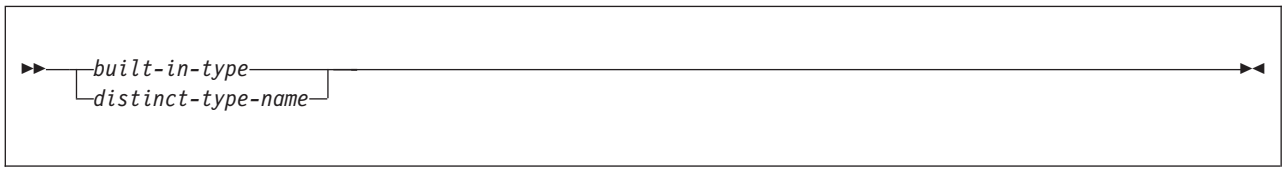


Notes:

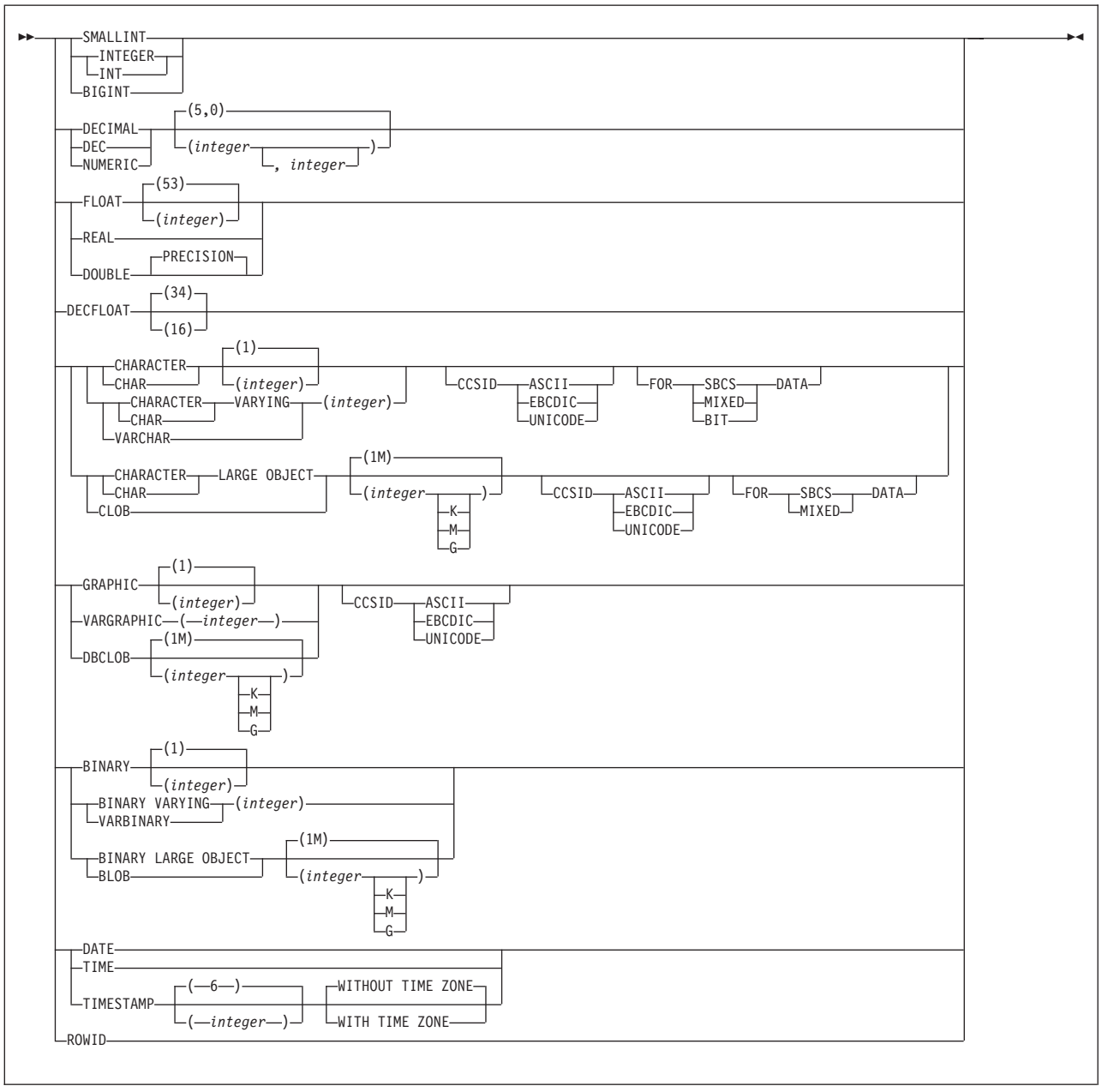
1 AS LOCATOR can be specified only for a LOB data type or a distinct type based on a LOB data type.

2 The TABLE LIKE name AS LOCATOR clause can only be specified for the parameter list of the function that is being defined.

data-type:



built-in-type:



Description

function-name

Names the user-defined function. The name is implicitly or explicitly qualified by a schema name.

The combination of name, schema name, the number of parameters, and the data type of each parameter[28] (without regard for any length, precision, scale, subtype or encoding scheme attributes of the data type) must not identify a user-defined function that exists at the current server.

If the function is sourced on an existing function to enable the use of the existing function with a distinct type, the name can be the same name as the existing function. In general, more than one function can have the same name if the function signature of each function is unique.

You can use the same name for more than one function if the function signature of each function is unique.

- The unqualified form of *function-name* must not be any of the following system-reserved keywords even if you specify them as delimited identifiers:

```
ALL                 LIKE                UNIQUE
AND                 MATCH               UNKNOWN
ANY                 NOT                 =
BETWEEN             NULL                ¬=
DISTINCT            ONLY                <
EXCEPT              OR                  <=
EXISTS              OVERLAPS            ¬<
FALSE               SIMILAR             >
FOR                 SOME                >=
FROM                TABLE               ¬>
IN                  TRUE                <>
IS                  TYPE
```

The schema name can be 'SYSTOOLS' or 'SYSFUN' if the user who executes the CREATE statement has SYSADM or SYSCTRL privilege. Otherwise, the schema name must not begin with 'SYS' unless the schema name is 'SYSADM'.

(*parameter-declaration***,...)**

Specifies the number of input parameters of the function and the data type of each parameter. All of the parameters for a function are input parameters and are nullable. There must be one entry in the list for each parameter that the function expects to receive. Although not required, you can give each parameter a name.

A function can have no parameters. In this case, you must code an empty set of parentheses, for example:

```
CREATE FUNCTION WOOFER()
```

parameter-name

Specifies the name of the input parameter. The name is an SQL identifier, and each name in the parameter list must not be the same as any other name.

data-type

Specifies the data type of the input parameter. The data type can be a built-in data type or a distinct type.

built-in-type

The data type of the input parameter is a built-in data type.

28. If the function has more than 30 parameters, only the first 30 parameters are used to determine whether the function is unique.

For information on the data types, see built-in-type.

For parameters with a character or graphic data type, the PARAMETER CCSID clause or CCSID clause indicates the encoding scheme of the parameter. If you do not specify either of these clauses, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

distinct-type-name
The data type of the input parameter is a distinct type. Any length, precision, scale, subtype, or encoding scheme attributes for the parameter are those of the source type of the distinct type.

The implicitly or explicitly specified encoding scheme of all of the parameters with a character or graphic string data type must be the same—either all ASCII, all EBCDIC, or all UNICODE.

Although parameters with a character data type have an implicitly or explicitly specified subtype (BIT, SBCS, or MIXED), the function program can receive character data of any subtype. Therefore, conversion of the input data to the subtype of the parameter might occur when the function is invoked.

Parameters with a datetime data type or a distinct type are passed to the function as a different data type:

- A datetime type parameter is passed as a character data type, and the data is passed in ISO format.

  The encoding scheme for a datetime type parameter is the same as the implicitly or explicitly specified encoding scheme of any character or graphic string parameters. If no character or graphic string parameters are passed, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.
- A distinct type parameter is passed as the source type of the distinct type.

You can specify any built-in data type or distinct type that matches or can be cast to the data type of the corresponding parameter of the source function (the function that is identified in the SOURCE clause). (For information on casting data types, see "Casting between data types" on page 111.) Length, precision, or scale attributes do not have be specified for data types with these attributes. When specifying data types with these attributes, follow these rules:

- An empty set of parentheses can be used to indicate that the length, precision, or scale is the same as the source function.
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default values are used.

AS LOCATOR
Specifies that a locator to the value of the parameter is passed to the function instead of the actual value. Specify AS LOCATOR only for parameters with a LOB data type or a distinct type based on a LOB data type. Passing locators instead of values can result in fewer bytes being passed to the function, especially when the value of the parameter is very large.

The AS LOCATOR clause has no effect on determining whether data types can be promoted, nor does it affect the function signature, which is used in function resolution.

TABLE LIKE *table-name* **or** *view-name* **AS LOCATOR**
Specifies that the parameter is a transition table. However, when the function is invoked, the actual values in the transition table are not passed to the function. A single value is passed instead. This single value is a locator to the table, which the function uses to access the columns of the transition table. A function with a table parameter can only be invoked from the triggered action of a trigger.

The use of TABLE LIKE provides an implicit definition of the transition table. It specifies that the transition table has the same number of columns as the identified table or view. If a table is specified, the transition table includes columns that are defined as implicitly hidden in the table. The columns have the same data type, length, precision, scale, subtype, and encoding scheme as the identified table or view, as they are described in catalog tables SYSCOLUMNS and SYSTABLESPACE. The number of columns and the attributes of those columns are determined at the time the CREATE FUNCTION statement is processed. Any subsequent changes to the number of columns in the table or the attributes of those columns do not affect the parameters of the function.

table-name or *view-name* must identify a table or view that exists at the current server. A view cannot have columns of length 0. The name must not identify a declared temporary table. The table that is identified can contain XML columns; however, the function cannot reference those XML columns. The name does not have to be the same name as the table that is associated with the transition table for the trigger. An unqualified table or view name is implicitly qualified according to the following rules:

- If the CREATE FUNCTION statement is embedded in a program, the implicit qualifier is the authorization ID in the QUALIFIER bind option when the plan or package was created or last rebound. If QUALIFIER was not used, the implicit qualifier is the owner of the plan or package.
- If the CREATE FUNCTION statement is dynamically prepared, the implicit qualifier is the SQL authorization ID in the CURRENT SCHEMA special register.

When the function is invoked, the corresponding columns of the transition table identified by the table locator and the table or view identified in the TABLE LIKE clause must have the same definition. The data type, length, precision, scale, and encoding scheme of these columns must match exactly. The description of the table or view at the time the CREATE FUNCTION statement was executed is used.

Additionally, a character FOR BIT DATA column of the transition table cannot be passed as input for a table parameter for which the corresponding column of the table specified at the definition is not defined as character FOR BIT DATA. (The definition occurs with the CREATE FUNCTION statement.) Likewise, a character column of the transition table that is not FOR BIT DATA cannot be passed as input for a table parameter for which the corresponding column of the table specified at the definition is defined as character FOR BIT DATA.

For more information about using table locators, see *DB2 Application Programming and SQL Guide*.

RETURNS
Identifies the output of the function.

data-type2
Specifies the data type of the output. The output is nullable.

You can specify any built-in data type or distinct type that can be cast from the data type of the result of the source function. (For information on casting data types, see "Casting between data types" on page 111.) For additional rules that apply to the data type that you can specify, see Rules for creating sourced functions.

AS LOCATOR
Specifies that the function returns a locator to the value rather than the actual value. You can specify AS LOCATOR only if the output from the function has a LOB data type or a distinct type based on a LOB data type.

SPECIFIC *specific-name*
Provides a unique name for the function. The name is implicitly or explicitly qualified with a schema name. The name, including the schema name, must not identify the specific name of another function that exists at the current server.

The unqualified form of *specific-name* is an SQL identifier. The qualified form is an SQL identifier (the schema name) followed by a period and an SQL identifier.

If you do not specify a schema name, it is the same as the explicit or implicit schema name of the function name (*function-name*). If you specify a schema name, it must be the same as the explicit or implicit schema name of the function name.

If you do not specify the SPECIFIC clause, the default specific name is the name of the function. However, if the function name does not provide a unique specific name or if the function name is a single asterisk, DB2 generates a specific name in the form of:

```
SQLxxxxxxxxxxxx
```

where `'xxxxxxxxxxxx'` is a string of 12 characters that make the name unique.

The specific name is stored in the SPECIFIC column of the SYSROUTINES catalog table. The specific name can be used to uniquely identify the function in several SQL statements (such as ALTER FUNCTION, COMMENT, DROP, GRANT, and REVOKE) and in DB2 commands (START FUNCTION, STOP FUNCTION, and DISPLAY FUNCTION). However, the function cannot be invoked by its specific name.

PARAMETER CCSID
Indicates whether the encoding scheme for character and graphic string parameters is ASCII, EBCDIC, or UNICODE. The default encoding scheme is the value specified in the CCSID clauses of the parameter list or RETURNS clause, or in the field DEF ENCODING SCHEME on installation panel DSNTIPF.

This clause provides a convenient way to specify the encoding scheme for character and graphic string parameters. If individual CCSID clauses are specified for individual parameters in addition to this PARAMETER CCSID clause, the value specified in *all* of the CCSID clauses must be the same value that is specified in this clause.

This clause also specifies the encoding scheme to be used for system-generated parameters of the routine such as message tokens and DBINFO.

SOURCE
Specifies that the new function is being defined as a sourced function. A *sourced function* is implemented by another function (the *source function*). The

source function must be a scalar or aggregate function that exists at the current server, and it must be one of the following types of functions:

- A function that was defined with a CREATE FUNCTION statement
- A cast function that was generated by a CREATE TYPE statement for a distinct type
- A built-in function

If the source function is not a built-in function, the particular function can be identified by its name, function signature, or specific name.

If the source function is a built-in function, the SOURCE clause must include a function signature for the built-in function. The source function must not be any of the built-in functions shown in Table 109 (if a particular syntax is shown, only the indicated form cannot be specified).

Table 109. Built-in functions that cannot be the source function. When listed with specific conditions, the function cannot be a source function under those conditions. Otherwise, the function cannot be a source function regardless of its arguments.

Type of function	Restricted functions
Aggregate	ARRAY_AGG COUNT(*) COUNT_BIG(*) XMLAGG

Table 109. Built-in functions that cannot be the source function (continued). When listed with specific conditions, the function cannot be a source function under those conditions. Otherwise, the function cannot be a source function regardless of its arguments.

Type of function	Restricted functions
Scalar function	ARRAY_DELETE ARRAY_FIRST ARRAY_LAST ARRAY_NEXT ARRAY_PRIOR CARDINALITY CHAR(*datetime-expression*, *second-argument*) where *second-argument* is ISO, USA, EUR, JIS, or LOCAL or if CHAR is specified with OCTETS, CODEUNITS16, or CODEUNITS32. CHARACTER_LENGTH CLOB if OCTETS, CODEUNITS16, or CODEUNITS32 is specified COALESCE if a parameter is an array DBCLOB if OCTETS, CODEUNITS16, or CODEUNITS32 is specified DECODE DECRYPT_BIT where second argument is DEFAULT DECRYPT_CHAR where second argument is DEFAULT DECRYPT_DB where second argument is DEFAULT EXTRACT GETVARIABLE where second argument is DEFAULT GRAPHIC if OCTETS, CODEUNITS16, or CODEUNITS32 is specified IFNULL if a parameter is an array INSERT if OCTETS, CODEUNITS16, or CODEUNITS32 is specified LEFT if OCTETS, CODEUNITS16, or CODEUNITS32 is specified LOCAL LOCATE if OCTETS, CODEUNITS16, or CODEUNITS32 is specified MAX MAX_CARDINALITY MIN NULLIF POSITION RID RIGHT if OCTETS, CODEUNITS16, or CODEUNITS32 is specified STRIP where multiple arguments are specified SUBSTRING TRIM where the first argument is BOTH, B, LEADING, L, TRAILING, T, or the first or second argument is FROM TRIM_ARRAY VARCHAR if OCTETS, CODEUNITS16, or CODEUNITS32 is specified VARGRAPHIC if OCTETS, CODEUNITS16, or CODEUNITS32 is specified XMLCONCAT XMLELEMENT XMLFOREST XMLNAMESPACES

If you base the sourced function directly or indirectly on an external scalar function, the sourced function inherits the attributes of the external scalar function. This can involve several layers of sourced functions. For example, assume that function A is sourced on function B, which in turn is sourced on function C. Function C is an external scalar function. Functions A and B inherit all of the attributes that are specified on the EXTERNAL clause of the CREATE FUNCTION statement for function C.

function-name
Identifies the function that is to be used as the source function. The source function can be defined with any number of parameters. If more than one function is defined with the specified name in the specified or implicit schema, an error is returned.

If you specify an unqualified *function-name*, DB2 searches the schemas of the SQL path. DB2 selects the first schema that has only one function with this name on which the user has EXECUTE authority. An error is returned if a function is not found or a schema has more than one function with this name.

function-name (parameter-type,...)
Identifies the function that is to be used as the source function by its function signature, which uniquely identifies the function. The*function-name (parameter-type,...)* must identify a function with the specified signature. The specified parameters must match the data types in the corresponding position that were specified when the function was created. DB2 uses the number of data types and the logical concatenation of the data types to identify the specific function instance. Synonyms for data types are considered a match.

If the function was defined with a table parameter (the LIKE TABLE *name* AS LOCATOR clause was specified in the CREATE FUNCTION statement to indicate that one of the input parameters is a transition table), the function signature cannot be used to uniquely identify the function. Instead, use one of the other syntax variations to identify the function with its function name, if unique, or its specific name.

If *function-name()* is specified, the identified function must have zero parameters.

function-name
Identifies the function name of the source function. If you specify an unqualified name, DB2 searches the schemas of the SQL path. Otherwise, DB2 searches for the function in the specified schema.

parameter-type,...
Identifies the parameters of the function.

If an unqualified distinct type name is specified, DB2 searches the SQL path to resolve the schema name for the distinct type.

Empty parentheses are allowed for some data types that are specified in this context. For data types that have a length, precision, or scale attribute, use one of the following specifications:

- Empty parentheses indicate that DB2 ignores the attribute when determining whether the data types match. For example, DEC() is considered a match for a parameter of a function that is defined with a data type of DEC(7,2). However, FLOAT cannot be specified with empty parentheses because its parameter value indicates a specific data type (REAL or DOUBLE).

- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement. If the data type is FLOAT, the precision does not need to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

If you omit the FOR *subtype* DATA clause or the CCSID clause for data types with a subtype or encoding scheme attribute, DB2 is to ignore the attribute when determining whether the data types match. An exception to ignoring the attribute is FOR BIT DATA. A character FOR BIT DATA parameter of the new function cannot correspond to a parameter of the source function that is not defined as character FOR BIT DATA. Likewise, a character parameter of the new function that is not FOR BIT DATA cannot correspond to a parameter of the source function that is defined as character FOR BIT DATA.

The number of input parameters in the function that is being created must be the same as the number of parameters in the source function. If the data type of each input parameter is not the same as or castable to the corresponding parameter of the source function, an error occurs. The data type of the final result of the source function must match or be castable to the result of the sourced function.

AS LOCATOR
Specifies that the function is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or distinct type that is based on a LOB.

SPECIFIC *specific-name*
Identifies the function to be used as the source function by its specific name.

If you specify an unqualified *specific-name*, DB2 searches the SQL path to locate the schema. DB2 selects the first schema that contains a function with this specific name for which the user has EXECUTE authority. DB2 returns an error if it cannot find a function with the specific name in one of the schemas in the SQL path.

If you specify a qualified *specific-name*, DB2 searches the named schema for the function. DB2 returns an error if it cannot find a function with the specific name.

Notes

Owner privileges:
The owner is authorized to execute the function (EXECUTE privilege) in the following cases:

- If the underlying function is a user-defined function, and the owner is authorized with the grant option to execute the underlying function, the privilege on the new function includes the grant option. Otherwise, the owner can execute the new function but cannot grant others the privilege to do so.

- If the underlying function is a built-in function, the owner is authorized with the grant option to execute the underlying built-in function and the privilege on the new function includes the grant option.

For more information, see "GRANT (function or procedure privileges)" on page 1703. For more information about ownership of the object, see "Authorization, privileges, permissions, masks, and object ownership" on page 70.

Choosing data types for parameters:

When you choose the data types of the input and output parameters for your function, consider the rules of promotion that can affect the values of the parameters. (See "Promotion of data types" on page 110). For example, a constant that is one of the input arguments to the function might have a built-in data type that is different from the data type that the function expects, and more significantly, might not be promotable to that expected data type. Based on the rules of promotion, consider using the following data types for parameters:

- INTEGER instead of SMALLINT
- DOUBLE instead of REAL
- VARCHAR instead of CHAR
- VARGRAPHIC instead of GRAPHIC
- VARBINARY instead of BINARY

For portability of functions across platforms that are not DB2 for z/OS, do not use the following data types, which might have different representations on different platforms:

- FLOAT. Use DOUBLE or REAL instead.
- NUMERIC. Use DECIMAL instead.

Specifying the encoding scheme for parameters:

The implicitly or explicitly specified encoding scheme of all of the parameters with a character or graphic string data type (both input and output parameters) must be the same—either all ASCII, all EBCDIC, or all UNICODE.

Determining the uniqueness of functions in a schema:

At the current server, the function signature of each function, which is the qualified function name combined with the number and data types of the input parameters, must be unique. If the function has more than 30 input parameters, only the data types of the first 30 are used to determine uniqueness. This means that two different schemas can each contain a function with the same name that have the same data types for all of their corresponding data types. However, a single schema must not contain multiple functions with the same name that have the same data types for all of their corresponding data types.

When determining whether corresponding data types match, DB2 does not consider any length, precision, or scale attributes in the comparison. DB2 considers the synonyms of data types as a match. For example, REAL and FLOAT, and DOUBLE and FLOAT are considered a match. Therefore, CHAR(8) and CHAR(35) are considered to be the same, as are DECIMAL(11,2), DECIMAL(4,3), DECFLOAT(16) and DECFLOAT(34), TIMESTAMP(6) and TIMESTAMP(9), TIMESTAMP(6) WITH TIME ZONE and TIMESTAMP(9) WITH TIME ZONE. Furthermore, the character and graphic types, and the timestamp types are considered to be the same. For example, the following are considered to be the same type: CHAR and

GRAPHIC, VARCHAR and VARGRAPHIC, CLOB and DBCLOB, TIMESTAMP WITHOUT TIME ZONE and TIMESTAMP WITH TIME ZONE. CHAR(13) and GRAPHIC(8) are considered to be the same type. An error is returned if the signature of the function being created is a duplicate of a signature for an existing user-defined function with the same name and schema.

Assume that the following statements are executed to create four functions in the same schema. The second and fourth statements fail because they create functions that are duplicates of the functions that the first and third statements created.

```
CREATE FUNCTION PART (INT, CHAR(15)) ...
CREATE FUNCTION PART (INTEGER, CHAR(40)) ...
CREATE FUNCTION ANGLE (DECIMAL(12,2)) ...
CREATE FUNCTION ANGLE (DEC(10,7)) ...
```

Rules for creating sourced functions:

Assume that the function that is being created is named NEWF and the source function is named SOURCEF. Consider the following rules when creating a sourced function:

- The unqualified names of the sourced function and source function can be different (NEWF and SOURCEF).
- The number of input parameters for NEWF and SOURCEF must be the same.
- When specifying the input parameters and output for NEWF, you can specify a value for the precision, scale, subtype, or encoding scheme for a data type with any of these attributes or use empty parentheses.

  Empty parentheses, such as VARCHAR(), indicate that the value of the attribute is the same as the attribute for the corresponding parameter of SOURCEF, or that is determined by data type promotion. If you specify any values for the attributes, DB2 checks the values against the corresponding input parameters and returned output of SOURCEF as described next.
- When the CREATE FUNCTION statement is executed, DB2 checks the input parameters of NEWF against those of SOURCEF. The data type of each input parameter of NEWF function must be either the same as, or promotable to, the data type of the corresponding parameter of SOURCEF. (For information on the promotion of data types, see "Casting between data types" on page 111.)

  This checking does not guarantee that an error will not occur when NEWF is invoked. For example, an argument that matches the data type and length or precision attributes of a NEWF parameter might not be promotable if the corresponding SOURCEF parameter has a shorter length or less precision. In general, do not define the parameters of a sourced function with length or precision attributes that are greater than the attributes of the corresponding parameters of the source function.
- When the CREATE FUNCTION statement is executed, DB2 checks the data type identified in the RETURNS clause of NEWF against the data type that SOURCEF returns. The data type that SOURCEF returns must be either the same as, or promotable to, the RETURNS data type of NEWF.

  This checking does not guarantee that an error will not occur when NEWF is invoked. For example, the value of a result that matches the data type and length or precision attributes of those specified for SOURCEF's result might not be promotable if the RETURNS data type

of NEWF has a shorter length or less precision. Consider the possible effects of defining the RETURNS data type of a sourced function with length or precision attributes that are less than the attributes defined for the data type returned by source function.

Scrollable cursors specified with user-defined functions:
A row can be fetched more than once with a scrollable cursor. Therefore, if a scrollable cursor is defined with a function that is not deterministic in the select list of the cursor, a row can be fetched multiple times with different results for each fetch. Similarly, if a scrollable cursor is defined with a user-defined function with external action, the action is executed with every fetch.

SECURED functions:
The sourced user-defined function inherits the SECURED or NOT SECURED attribute from the source function in which only the topmost user-defined function is considered. If the topmost user-defined function is secure, any nested user-defined functions are also considered secure. DB2 does not validate whether those nested user-defined functions are secure. If those nested functions can access sensitive data, the security administrator needs to ensure that those functions are allowed to access sensitive data and should ensure that a change control audit procedure has been established for all changes to those functions.

If the sourced function is using the VERIFY_GROUP_FOR_USER or VERIFY_ ROLE_FOR_USER function as its source, the sourced function must specify only two input parameters.

Functions and global variables:
The content of global variables that are referenced in functions is inherited from the caller.

If the function references global variables, the level of SQL data access must be at least CONTAINS SQL. If the function contains SQL statements that make modifications to global variables, the level of SQL data access must be MODIFIES SQL DATA.

Examples

Example 1: Assume that you created a distinct type HATSIZE, which you based on the built-in data type INTEGER. You want to have an AVG function to compute the average hat size of different departments. Create a sourced function that is based on built-in function AVG.

```
CREATE FUNCTION AVE (HATSIZE) RETURNS HATSIZE
   SOURCE SYSIBM.AVG (INTEGER);
```

When you created distinct type HATSIZE, two cast functions were generated, which allow HATSIZE to be cast to INTEGER for the argument and INTEGER to be cast to HATSIZE for the result of the function.

Example 2: After Smith registered the external scalar function CENTER in his schema, you decide that you want to use this function, but you want it to accept two INTEGER arguments instead of one INTEGER argument and one FLOAT argument. Create a sourced function that is based on CENTER.

```
CREATE FUNCTION MYCENTER (INTEGER, INTEGER)
   RETURNS FLOAT
   SOURCE SMITH.CENTER (INTEGER, FLOAT);
```

CREATE FUNCTION (SQL scalar)

The CREATE FUNCTION (SQL Scalar) statement defines an SQL scalar function at the current server and specifies the source statements for the function. The body of the function is written in the SQL procedural language. The function returns a single value each time it is invoked.

If the function that is created with the CREATE FUNCTION (SQL scalar) statement could have been defined prior to DB2 Version 10 for z/OS, the function is an inline SQL scalar function. No package will be created for an inline SQL scalar function. However, if the function could have been defined prior to DB2 Version 10 for z/OS, except that an input parameter or the RETURNS parameter uses the XML data type, the function will still be considered an inline SQL scalar function.

For non-inline SQL scalar functions, you can define multiple versions of the function. Use CREATE FUNCTION to define the initial version, and ALTER FUNCTION to define subsequent versions. For information about the SQL control statements that are supported in SQL functions, refer to Chapter 6, "SQL control statements for SQL routines," on page 1963.

Invocation

For an inline SQL function, this statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

For a non-inline SQL function, this statement can only be dynamically prepared but the DYNAMICRULES run behavior must be specified implicitly or explicitly.

Authorization

The privilege set defined below must include at least one of the following:

- The CREATEIN privilege on the schema and the required authorization to add a new package or a new version of an existing package, depending on the value of the BIND NEW PACKAGE field on installation panel DSNTIPP
- SYSADM or SYSCTRL authority
- System DBADM

The authorization ID that matches the schema name implicitly has the CREATEIN privilege on the schema.

If the authorization ID that is used to create the function has installation SYSADM authority, the function is identified as system-defined function.

If a user-defined type is referenced (as the data type of a parameter or an SQL variable), the privilege set must also include at least one of the following:

- Ownership of the user-defined type
- The USAGE privilege on the user-defined type
- SYSADM authority

If the function uses a table as a parameter, the privilege set must also include at least one of the following:

- Ownership of the table

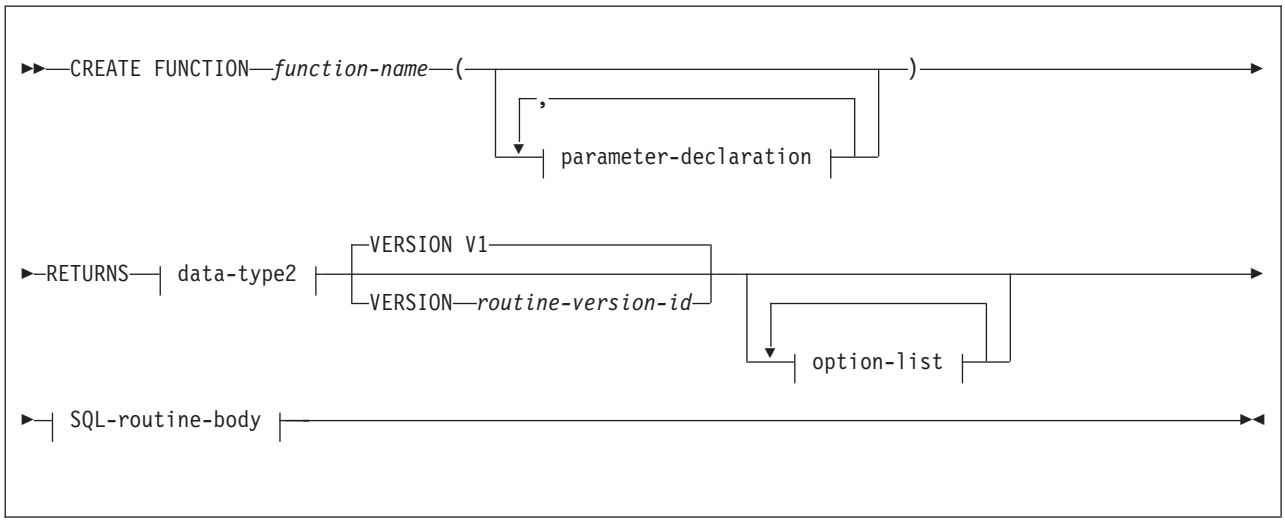
- The SELECT privilege on the table
- SYSADM authority

At least one of the following additional privileges is required if the SECURED option is specified
- SECADM authority
- CREATE_SECURE_OBJECT privilege

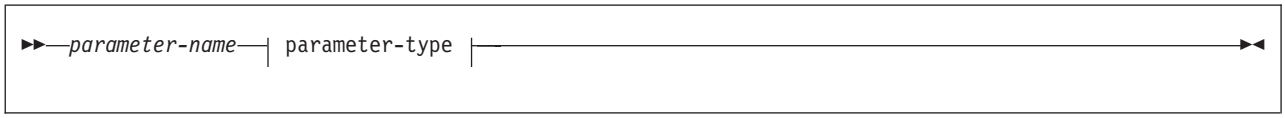
Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the owner is a role, the implicit schema match does not apply and this role needs to include one of the previously listed conditions.

If the statement is dynamically prepared and is not running in a trusted context for which the ROLE AS OBJECT OWNER clause is specified, the privilege set is the set of privileges that are held by the SQL authorization ID of the process. If the schema name is not the same as the SQL authorization ID of the process, one of the following conditions must be met:
- The privilege set includes SYSADM or SYSCTRL authority.
- The SQL authorization ID of the process has the CREATEIN privilege on the schema.

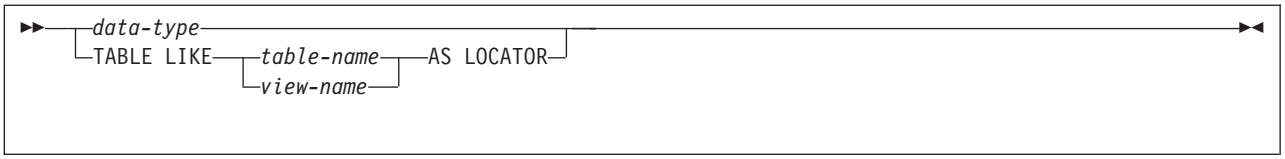
Syntax



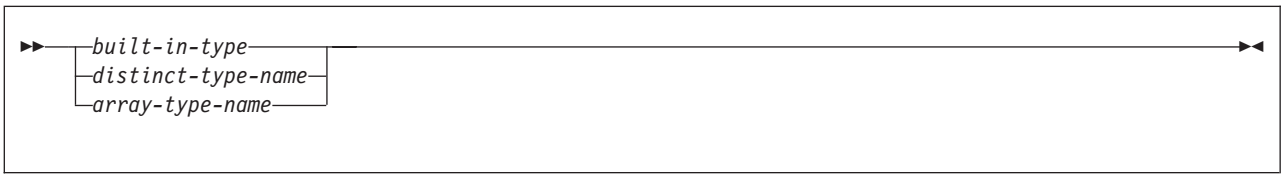
parameter-declaration:



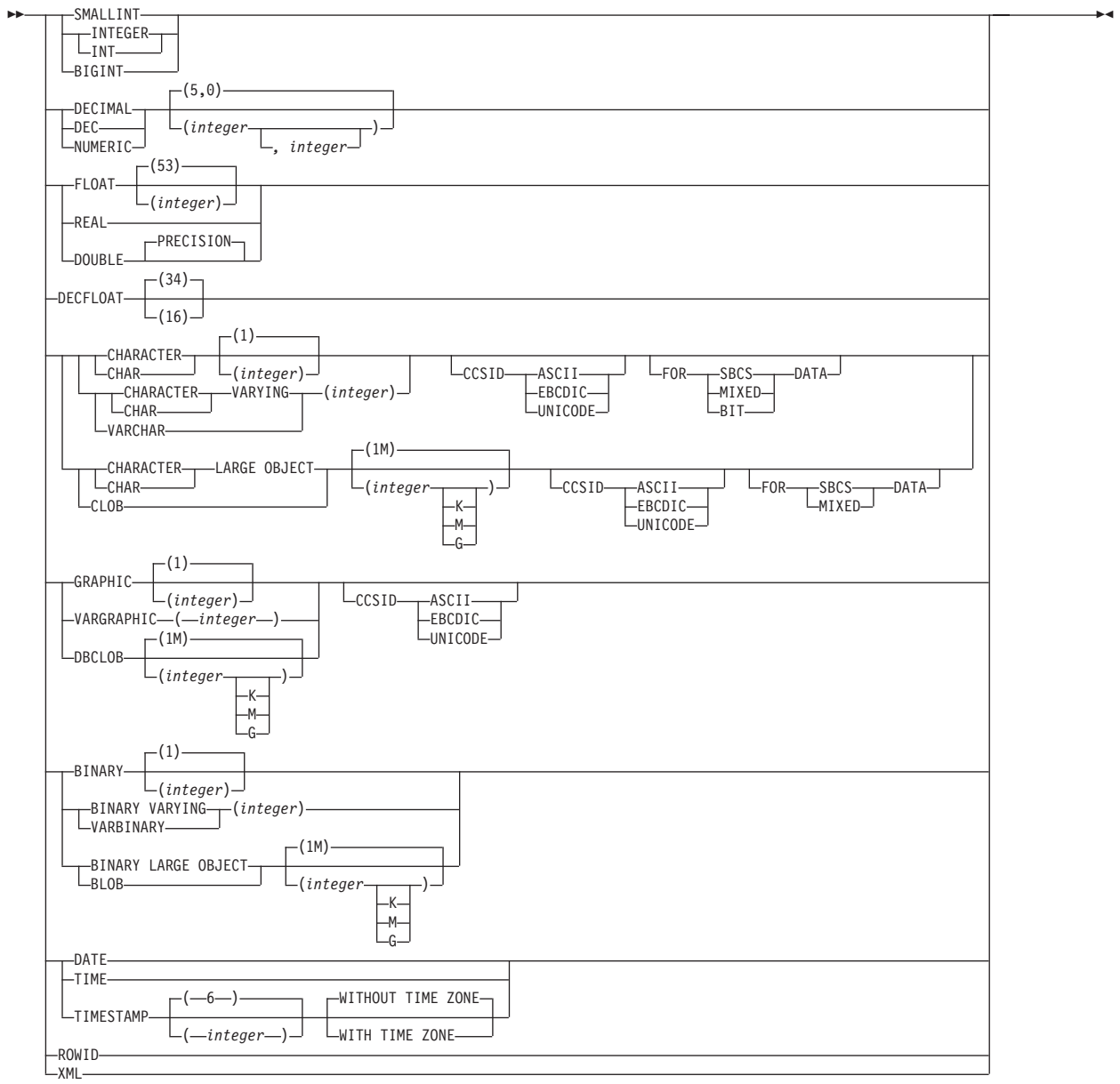
parameter-type:


data-type
TABLE LIKE
table-name
view-name
AS LOCATOR

data-type:

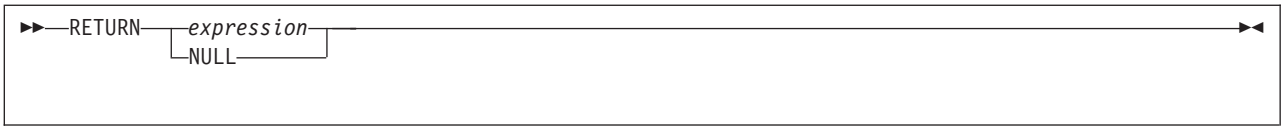

built-in-type
distinct-type-name
array-type-name

built-in-type:

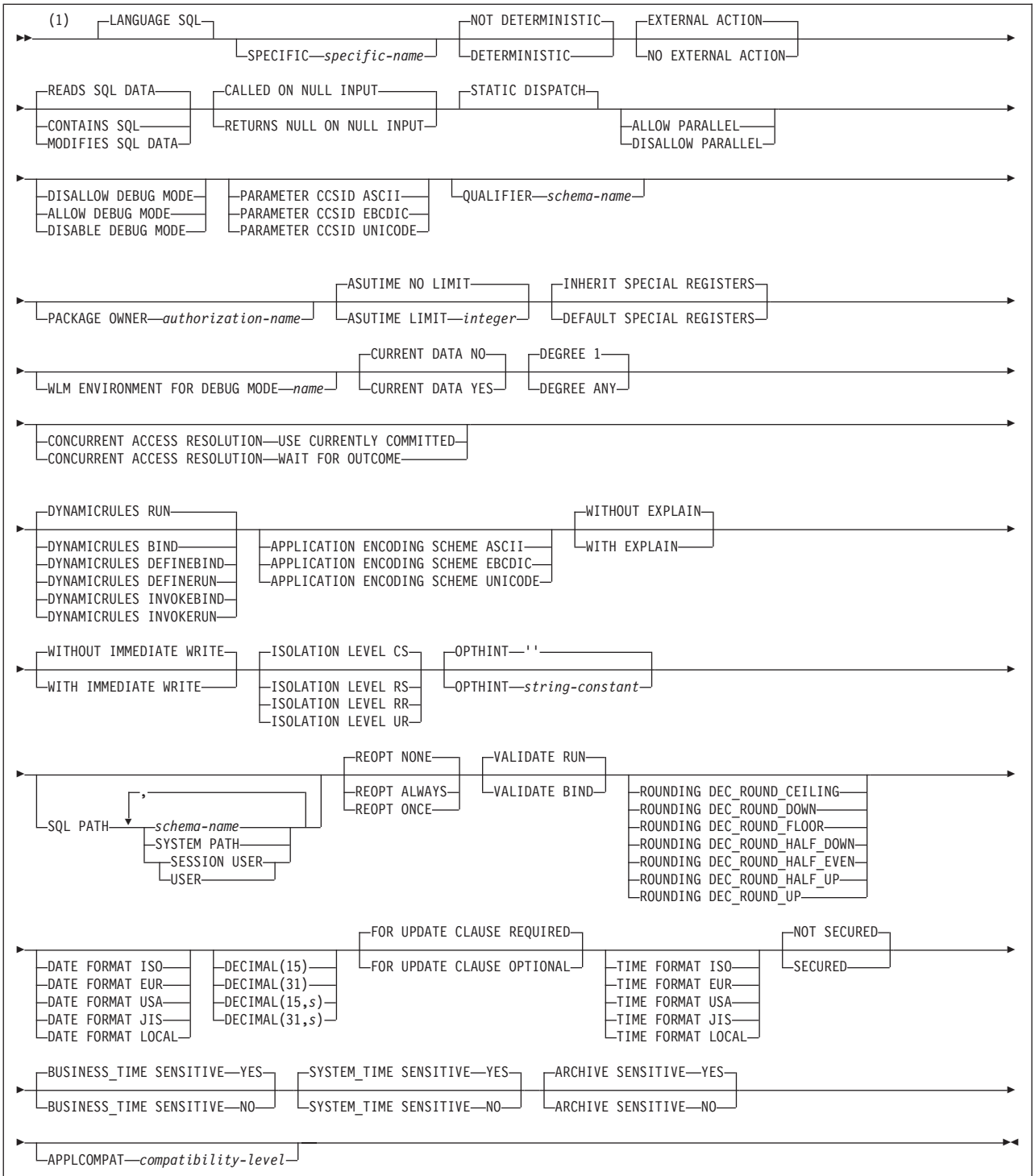


SQL-routine-body:

simple-return-statement:



option-list: (The options in the *option-list* can be specified in any order, but each one can only be specified one time)



Notes:

1 Only LANGUAGE SQL, SPECIFIC, PARAMETER CCSID, NOT DETERMINISTIC, DETERMINISTIC, EXTERNAL ACTION, NO EXTERNAL ACTION, CONTAINS SQL, READS SQL DATA, STATIC DISPATCH, CALLED ON NULL INPUT, NOT SECURED, and SECURED can be specified for an inline SQL scalar function. If other options are specified, the function will be considered a non-inline SQL scalar function.

Description

function-name
Names the function. If *function-name* already exists with the specified signature, an error is returned even if VERSION is specified with a *routine-version-id* that is different from any existing version identifier for the function that is specified in *function-name*.[29]

(*parameter-declaration***,...)**

Specifies the number of input parameters of the function and the name and data type of each parameter. Each *parameter-declaration* specifies an input parameter for the function. A function can have zero or more input parameters. There must be one entry in the list for each parameter that the function expects to receive. All of the parameters for a function are input parameters and are nullable. If the function has more than 30 parameters, only the first 30 parameters are used to determine if the function is unique.

parameter-name
Specifies the name of the input parameter. The name is an SQL identifier, and each name in the parameter list must not be the same as any other name.

data-type
Specifies the data type of the input parameter. The data type can be a built-in data type or a user-defined type.

built-in-type
The data type of the input parameter is a built-in data type.

For information on the data types, see built-in-type.

For parameters with a character or graphic data type, the PARAMETER CCSID clause or CCSID clause indicates the encoding scheme of the parameter. If you do not specify either of these clauses, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

distinct-type-name
The data type of the input parameter is a distinct type. Any length, precision, scale, subtype, or encoding scheme attributes for the parameter are those of the source type of the distinct type. The distinct type must not be based on a LOB data type.

If you specify the name of the distinct type without a schema name, DB2 resolves the distinct type by searching the schemas in the SQL path.

array-type-name
The data type of the input parameter is a user-defined array type.

If you specify *array-type-name* without a schema name, DB2 resolves the array type by searching the schemas in the SQL path.

The implicitly or explicitly specified encoding scheme of all of the parameters with a character or graphic string data type must be the same—either all ASCII, all EBCDIC, or all UNICODE.

Although parameters with a character data type have an implicitly or explicitly specified subtype (BIT, SBCS, or MIXED), the function program

29. If the function has more than 30 parameters, only the first 30 parameters are used to determine whether the function is unique.

can receive character data of any subtype. Therefore, conversion of the input data to the subtype of the parameter might occur when the function is invoked. An error occurs if mixed data that actually contains DBCS characters is used as the value for an input parameter that is declared with an SBCS subtype.

Parameters with a datetime data type or a distinct type are passed to the function as a different data type:

- A datetime type parameter is passed as a character data type, and the data is passed in ISO format.

  The encoding scheme for a datetime type parameter is the same as the implicitly or explicitly specified encoding scheme of any character or graphic string parameters. If no character or graphic string parameters are passed, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.
- A distinct type parameter is passed as the source type of the distinct type.

RETURNS

Identifies the output of the function.

data-type2

Specifies the data type of the output. The output is nullable.

The same considerations that apply to the data type of input parameter, as described under data-type, apply to the data type of the output of the function.

VERSION *routine-version-id*

Specifies the version identifier for the first version of the function that is to be generated. You can use an ALTER FUNCTION statement with the ADD VERSION clause or the BIND DEPLOY command to create additional versions of the function.

routine-version-id

An SQL identifier of up to 64 EBCDIC bytes that designates a version of a routine. The UTF-8 representation of the name must not exceed 122 bytes.

V1 is the default version identifier.

SPECIFIC *specific-name*

Specifies a unique name for the function. The name is implicitly or explicitly qualified with a schema name. The name, including the schema name, must not identify the specific name of another function that exists at the current server.

The unqualified form of *specific-name* is an SQL identifier. The qualified form is an SQL identifier (the schema name) followed by a period and an SQL identifier.

If you do not specify a schema name, it is the same as the explicit or implicit schema name of the function name (*function-name*). If you specify a schema name, it must be the same as the explicit or implicit schema name of the function name.

If you do not specify the SPECIFIC clause, the default specific name is the name of the function. However, if the function name does not provide a unique specific name or if the function name is a single asterisk, DB2 generates a specific name in the form of:

```
SQLxxxxxxxxxxxx
```

where 'xxxxxxxxxxxx' is a string of 12 characters that make the name unique.

The specific name is stored in the SPECIFIC column of the SYSROUTINES catalog table. The specific name can be used to uniquely identify the function in several SQL statements (such as ALTER FUNCTION, COMMENT, DROP, GRANT, and REVOKE) and must be used in DB2 commands (START FUNCTION, STOP FUNCTION, and DISPLAY FUNCTION). However, the function cannot be invoked by its specific name.

RETURN
Specifies a simple form of the RETURN statement that will not result in the creation of a package when RETURN is the only statement in the function body. The RETURN statement specifies the result of the function.

expression
The data type of the result must be assignable to the data type that is defined for the function result, using the storage assignment rules as described in "Assignment and comparison" on page 121. An aggregate function, a user-defined function that is sourced on an aggregate function, or an OLAP specification must not be specified for the RETURN statement. See "Expressions" on page 240 for information on expressions. For this simple form of the RETURN statement, *expression* cannot contain a scalar fullselect. If *expression* contains a fullselect, the function is a non-inline function, a package is created, and the RETURN statement is subject to the rules specified in "RETURN statement" on page 2003.

NULL
Specifies that the null value is returned.

NOT DETERMINISTIC or DETERMINISTIC
Specifies whether the function returns the same results each time that the function is invoked with the same input arguments.

NOT DETERMINISTIC
The function might not return the same result each time that the function is invoked with the same input arguments. The function depends on some state values that affect the results. DB2 uses this information to disable the merging of views and table expressions when processing SELECT and SQL data change statements that refer to this function. An example of a function that is not deterministic is one that generates random numbers.

NOT DETERMINISTIC must be specified explicitly or implicitly if the function program accesses a special register or invokes another function that is not deterministic. NOT DETERMINISTIC is the default.

DETERMINISTIC
The function always returns the same result function each time that the function is invoked with the same input arguments. An example of a deterministic function is a function that calculates the square root of the input. DB2 uses this information to enable the merging of views and table expressions for SELECT and SQL data change statements that refer to this function. DETERMINISTIC is not the default. If applicable, specify DETERMINISTIC to prevent non-optimal access paths from being chosen for SQL statements that refer to this function.

DB2 does not verify that the function program is consistent with the specification of DETERMINISTIC or NOT DETERMINISTIC.

EXTERNAL ACTION or NO EXTERNAL ACTION

Specifies whether the function takes an action that changes the state of an object that DB2 does not manage. An example of an external action is sending a message or writing a record to a file.

EXTERNAL ACTION

The function can take an action that changes the state of an object that DB2 does not manage.

Some SQL statements that invoke functions with external actions can result in incorrect results if parallel tasks execute the function. For example, if the function sends a note for each initial call to it, one note is sent for each parallel task instead of once for the function. Specify the DISALLOW PARALLEL clause for functions that do not work correctly with parallelism.

If you specify EXTERNAL ACTION, then DB2:

- Materializes the views and table expressions in SELECT and SQL data change statements that refer to the function. This materialization can adversely affect the access paths that are chosen for the SQL statements that refer to this function. Do not specify EXTERNAL ACTION if the function does not have an external action.
- Does not move the function from one task control block (TCB) to another between FETCH operations.
- Does not allow another function or stored procedure to use the TCB until the cursor is closed. This is also applicable for cursors declared WITH HOLD.

The only changes to resources made outside of DB2 that are under the control of commit and rollback operations are those changes made under RRS control.

EXTERNAL ACTION must be specified implicitly or explicitly specified if the SQL routine body invokes a function that is defined with EXTERNAL ACTION. EXTERNAL ACTION is the default.

NO EXTERNAL ACTION

The function does not take any action that changes the state of an object that DB2 does not manage. DB2 uses this information to enable the merging of views and table expressions for SELECT and SQL data change statements that refer to this function. If applicable, specify NO EXTERNAL ACTION to prevent non-optimal access paths from being chosen for SQL statements that refer to this function.

Although the scope of global variables are beyond the scope of the routine, global variables can be set in the routine body when NO EXTERNAL ACTION is specified.

DB2 does not verify that the function program is consistent with the specification of EXTERNAL ACTION or NO EXTERNAL ACTION.

MODIFIES SQL DATA, READS SQL DATA, or CONTAINS SQL

Specifies which SQL statements, if any, can be executed in the function or any routine that is called from the function. For the data access classification of each statement, see Table 162 on page 2030.

MODIFIES SQL DATA

Specifies that the function can execute any SQL statement except the statements that are not supported in functions. Do not specify MODIFIES SQL DATA when ALLOW PARALLEL is in effect.

READS SQL DATA
Specifies that the function can execute statements with a data access classification of READS SQL DATA, CONTAINS SQL, or NO SQL. The function cannot execute SQL statements that modify data.

READS SQL DATA is the default.

CONTAINS SQL
Specifies that the function can execute only SQL statements with a data access classification of CONTAINS SQL or NO SQL. The function cannot execute SQL statements that read or modify data.

CALLED ON NULL INPUT or RETURNS NULL ON NULL INPUT
Specifies whether the function is invoked if any of the input arguments is null at execution time.

CALLED ON NULL INPUT
Specifies that the function is to be invoked if any, or if all, of the argument values are null. Specifying CALLED ON NULL INPUT means that the body of the function must be coded to test for null argument values.

CALLED ON NULL INPUT is the default.

RETURNS NULL ON NULL INPUT
Specifies that the function is not invoked and returns the null value if any of the input argument values is null.

STATIC DISPATCH
At function resolution time, DB2 chooses a function based on the static (or declared) types of the function parameters. **STATIC DISPATCH** is the default.

ALLOW PARALLEL or DISALLOW PARALLEL
Specifies if the function can be run in parallel. The default is DISALLOW PARALLEL, if you specify one or more of the following clauses:

- NOT DETERMINISTIC
- EXTERNAL ACTION
- MODIFIES SQL DATA

Otherwise, ALLOW PARALLEL is the default.

ALLOW PARALLEL
Specifies that the function can be run in parallel.

DISALLOW PARALLEL
Specifies that the function cannot be run in parallel.

ALLOW DEBUG MODE, DISALLOW DEBUG MODE, or DISABLE DEBUG MODE
Specifies whether this version of the routine can be run in debugging mode. The default is determined using the value of the CURRENT DEBUG MODE special register.

ALLOW DEBUG MODE
Specifies that this version of the routine can be run in debugging mode. When this version of the routine is invoked and debugging is attempted, a WLM environment must be available.

DISALLOW DEBUG MODE
Specifies that this version of the routine cannot be run in debugging mode.

You can use an ALTER statement to change this option to ALLOW DEBUG MODE for this initial version of the routine.

DISABLE DEBUG MODE
Specifies that this version of the routine can never be run in debugging mode.

This version of the routine cannot be changed to specify ALLOW DEBUG MODE or DISALLOW DEBUG MODE after this version of the routine has been created or altered to use DISABLE DEBUG MODE. To change this option, drop the routine and create it again using the option that you want. An alternative to dropping and recreating the routine is to create a version of the routine that uses the option that you want and making that version the active version.

When DISABLE DEBUG MODE is in effect, the WLM ENVIRONMENT FOR DEBUG MODE is ignored.

PARAMETER CCSID
Indicates whether the encoding scheme for character or graphic string parameters is ASCII, EBCDIC, or UNICODE. The default encoding scheme is the value that is specified in the CCSID clauses of the parameter list or in the field DEF ENCODING SCHEME on installation panel DSNTIPF.

This clause provides a convenient way to specify the encoding scheme for character or graphic string parameters. If individual CCSID clauses are specified for individual parameters in addition to this PARAMETER CCSID clause, the value that is specified in *all* of the CCSID clauses must be the same value that is specified in this clause.

If the data type for a parameter is a user-defined distinct type that is defined as a character or graphic type string, the CCSID of the distinct type must be the same as the value that is specified in this clause.

If the data type for a parameter is a user-defined array type that is defined with character or graphic string array elements, or a character string array index, the CCSID of these array attributes must be the same as the value that is specified in this clause.

This clause also specifies the encoding scheme that will be used for system-generated parameters of the routine.

QUALIFIER *schema-name*
Specifies the implicit qualifier that is used for unqualified names of tables, views, indexes, and aliases that are referenced in the routine body. The default value is the same as the default schema.

PACKAGE OWNER *authorization-name*
Specifies the owner of the package that is associated with the first version of the routine. The SQL authorization ID of the process is the default value.

The authorization ID must have the privileges that are required to execute the SQL statements that are contained in the routine body and must contain the necessary bind privileges. The value of PACKAGE OWNER is subject to translation when it is sent to a remote system.

If the privilege set lacks SYSADM or SYSCTRL authority, *authorization-name* must be the same as one of the authorization IDs of the process or the authorization ID of the process. If the privilege set includes SYSADM or SYSCTRL authority, *authorization-name* can be any authorization ID that contains the necessary bind privileges.

ASUTIME
Specifies the total amount of processor time, in CPU service units, that a single

invocation of a routine can run. The value is unrelated to the ASUTIME column of the resource limit specification table.

When you are debugging a routine, setting a limit can be helpful in case the routine gets caught in a loop. For information on service units, see *z/OS MVS Initialization and Tuning Guide*.

NO LIMIT
Specifies that there is no limit on the service units.

NO LIMIT is the default.

LIMIT *integer*
The limit on the number of CPU service units is a positive *integer* in the range of 1 to 2 147 483 647. If the procedure uses more service units than the specified value, DB2 cancels the procedure. The CPU cycles that are consumed by parallel tasks in a procedure do not contribute towards the specified ASUTIME LIMIT.

INHERIT SPECIAL REGISTERS or DEFAULT SPECIAL REGISTERS
Specifies how special registers are set on entry to the routine.

INHERIT SPECIAL REGISTERS
Specifies that the values of special registers are inherited, according to the rules that are listed in the table for characteristics of special registers in a routine in Table 40 on page 205.

INHERIT SPECIAL REGISTERS is the default.

DEFAULT SPECIAL REGISTERS
Specifies that special registers are initialized to the default values, as indicated by the rules in the table for characteristics of special registers in a routine in Table 40 on page 205.

WLM ENVIRONMENT FOR DEBUG MODE *name*
Specifies the WLM (workload manager) application environment that is used by DB2 when debugging the routine. The *name* of the WLM environment is an SQL identifier.

If you do not specify WLM ENVIRONMENT FOR DEBUG MODE, DB2 uses the default WLM-established stored procedure address space specified at installation time.

To define a routine that is to run in a specified WLM application environment, you must have the appropriate authority for the WLM application environment. For an example of a RACF command that provides this authorization, see Running stored procedures.

The WLM ENVIRONMENT FOR DEBUG MODE value is ignored when DISABLE DEBUG MODE is in effect.

CURRENT DATA(YES) or CURRENT DATA(NO)
Specifies whether to require data currency for read-only and ambiguous cursors when the isolation level of cursor stability is in effect. CURRENT DATA also determines whether block fetch can be used for distributed, ambiguous cursors.

YES
Specifies that data currency is required for read-only and ambiguous cursors. DB2 acquired page or row locks to ensure data currency. Block fetch is ignored for distributed, ambiguous cursors.

NO Specifies that data currency is not required for read-only and ambiguous

cursors. Block fetch is allowed for distributed, ambiguous cursors. Use of **CURRENT DATA**(**NO**) is not recommended if the routine attempts to dynamically prepare and execute a DELETE WHERE CURRENT OF statement against an ambiguous cursor after that cursor is opened. You receive a negative SQLCODE if your routine attempts to use a DELETE WHERE CURRENT OF statement for any of the following cursors:

- A cursor that is using block fetch
- A cursor that is using query parallelism
- A cursor that is positioned on a row that is modified by this or another application process

NO is the default.

DEGREE

Specifies whether to attempt to run a query using parallel processing to maximize performance.

1 Specifies that parallel processing should not be used.

1 is the default.

ANY

Specifies that parallel processing can be used.

CONCURRENT ACCESS RESOLUTION

Specifies the whether processing uses only committed data or whether it will wait for commit or rollback of data that is in the process of being updated.

WAIT FOR OUTCOME

Specifies that processing will wait for the commit or rollback of data that is in the process of being updated.

USE CURRENTLY COMMITTED

Specifies that processing use the currently committed version of the data when data that is in the process of being updated is encountered. **USE CURRENTLY COMMITTED** is applicable on scans that access tables that are defined in universal table spaces with row or page level lock size.

When there is lock contention between a read transaction and an insert transaction, **USE CURRENTLY COMMITTED** is applicable to scans with isolation level CS or RS. Applicable scans include intent read scans for read-only and ambiguous queries and for updatable cursors. **USE CURRENTLY COMMITTED** is also applicable to scans initiated from WHERE predicates of UPDATE or DELETE statements and the subselect of INSERT statements.

When there is lock contention is between a read transaction and a delete transaction, **USE CURRENTLY COMMITTED** is applicable to scans with isolation level CS and when CURRENTDATA(NO) is specified.

DYNAMICRULES

Specifies the values that apply, at run time, for the following dynamic SQL attributes:

- The authorization ID that is used to check authorization
- The qualifier that is used for unqualified objects
- The source for application programming options that DB2 uses to parse and semantically verify dynamic SQL statements

DYNAMICRULES also specifies whether dynamic SQL statements can include GRANT, REVOKE, ALTER, CREATE, DROP, and RENAME statements.

In addition to the value of the DYNAMICRULES clause, the run time environment of a routine controls how dynamic SQL statements behave at run time. The combination of the DYNAMICRULES value and the run time environment determines the value for the dynamic SQL attributes. That set of attribute values is called the dynamic SQL statement behavior. The following values can be specified:

RUN
Specifies that dynamic SQL statements are to be processed using run behavior.

RUN is the default.

BIND
Specifies that dynamic SQL statements are to be processed using bind behavior.

DEFINEBIND
Specifies that dynamic SQL statements are to be processed using either define behavior or bind behavior.

DEFINERUN
Specifies that dynamic SQL statements are to be processed using either define behavior or run behavior.

INVOKEBIND
Specifies that dynamic SQL statements are to be processed using either invoke behavior or bind behavior.

INVOKERUN
Specifies that dynamic SQL statements are to be processed using either invoke behavior or run behavior.

See "Authorization IDs and dynamic SQL" on page 75 for information on the effects of these options.

APPLICATION ENCODING SCHEME
Specifies the default encoding scheme for SQL variables in static SQL statements in the routine body. The value is used for defining an SQL variable in a compound statement if the CCSID clause is not specified as part of the data type, and the PARAMETER CCSID routine option is not specified.

ASCII
Specifies that the data is encoded using the ASCII CCSIDs of the server.

EBCDIC
Specifies that the data is encoded using the EBCDIC CCSIDs of the server.

UNICODE
Specifies that the data is encoded using the Unicode CCSIDs of the server.

See the ENCODING bind option in *DB2 Command Reference* for information about how the default for this option is determined.

WITH EXPLAIN or WITHOUT EXPLAIN
Specifies whether information will be provided about how SQL statements in the routine will execute.

WITHOUT EXPLAIN
Specifies that information will not be provided about how SQL statements in the routine will execute.

You can get EXPLAIN output for a statement that is embedded in a routine that is specified using WITHOUT EXPLAIN by embedding the SQL statement EXPLAIN in the routine body. Otherwise, the value of the EXPLAIN option applies to all explainable SQL statements in the routine body, and to the fullselect portion of any DECLARE CURSOR statements.

WITHOUT EXPLAIN is the default.

WITH EXPLAIN
Specifies that information will be provided about how SQL statements in the routine will execute. Information is inserted into the table *owner*.PLAN_TABLE. *owner* is the authorization ID of the owner of the routine. Alternatively, the authorization ID of the owner of the routine can have an alias as *owner*.PLAN_TABLE that points to the base table, PLAN_TABLE. *owner* must also have the appropriate SELECT and INSERT privileges on that table. WITH EXPLAIN does not obtain information for statements that access remote objects. PLAN_TABLE must have a base table and can have multiple aliases with the same table name, PLAN_TABLE, but have different schema qualifiers. It cannot be a view or a synonym and should exist before the version is added or replaced. In all inserts to *owner*.PLAN_TABLE, the value of QUERYNO is the statement number that is assigned by DB2.

The WITH EXPLAIN option also populates two optional tables if they exist: DSN_STATEMNT_TABLE and DSN_FUNCTION_TABLE. DSN_STATEMNT_TABLE contains an estimate of the processing cost for an SQL statement. See *DB2 Application Programming and SQL Guide* for more information. DSN_FUNCTION_TABLE contains information about function resolution. See *DB2 Application Programming and SQL Guide* for more information.

For a description of the tables that are populated by the WITH EXPLAIN option, see "EXPLAIN" on page 1642.

WITH IMMEDIATE WRITE or WITHOUT IMMEDIATE WRITE
Specifies whether immediate writes are to be done for updates that are made to group buffer pool dependent page sets or partitions. This option is only applicable for data sharing environments. The IMMEDWRITE subsystem parameter has no affect of this option. *DB2 Command Reference* shows the implied hierarchy of the IMMEDWRITE bind option (which is similar to this routine option) as it affects run time.

WITHOUT IMMEDIATE WRITE
Specifies that normal write activity is performed. Updated pages that are group buffer pool dependent are written at or before phase one of commit or at the end of abort for transactions that have been rolled back.

WITHOUT IMMEDIATE WRITE is the default.

WITH IMMEDIATE WRITE
Specifies that updated pages that are group buffer pool dependent are immediately written as soon as the buffer update completes. Updated pages are written immediately even if the buffer is updated during forward progress or during the rollback of a transaction. **WITH IMMEDIATE WRITE** might impact performance.

ISOLATION LEVEL RR, RS, CS, or UR
Specifies how far to isolate the routine from the effects of other running applications. For information about isolation levels, see *DB2 Performance Monitoring and Tuning Guide*.

RR Specifies repeatable read.

RS Specifies read stability.

CS Specifies cursor stability. **CS** is the default.

UR Specifies uncommitted read.

OPTHINT *string-constant*
Specifies whether query optimization hints are used for static SQL statements that are contained within the body of the routine.

string-constant is a character string of up to 128 bytes in length, which is used by the DB2 subsystem when searching the PLAN_TABLE for rows to use as input. The default value is an empty string, which indicates that the DB2 subsystem does not use optimization hints for static SQL statements.

Optimization hints are only used if optimization hints are enabled for your system. See *DB2 Installation Guide* for information about enabling optimization hints.

SQL PATH
Specifies the SQL path that the DB2 subsystem uses to resolve unqualified user-defined data types, functions, and procedure names (in CALL statements) in the body of the routine. The maximum length of the SQL path is 2048 bytes. DB2 calculates the length by taking each *schema-name* that is specified and removing any trailing blanks from it, adding a delimiter on the left and right sides, and adding one comma after each schema name except for the last name. The length of the resulting string cannot exceed 2048 bytes.

schema-name
Identifies a schema. DB2 does not verify that the schema exists when the CREATE statement is processed. The same schema name should not appear more than one time in the list of schema names.

SYSPUBLIC must not be specified for the SQL path.

SYSTEM PATH
Specifies the schema names "SYSIBM", "SYSFUN", "SYSPROC", "SYSIBMADM".

SESSION_USER or USER
Specifies the value of the SESSION_USER (or USER) special register. At the time the CREATE statement is processed, the actual length is included in the total length of the list of schema names that is specified for the SQL PATH option.

REOPT
Specifies if DB2 will determine the access path at run time by using the values of SQL variables or SQL parameters, parameter markers, and special registers.

NONE
Specifies that DB2 does not determine the access path at run time by using the values of SQL variables or SQL parameters, parameter markers, and special registers.

NONE is the default.

ALWAYS
Specifies that DB2 always determines the access path at run time each time an SQL statement is run. Do not specify REOPT ALWAYS with the WITH KEEP DYNAMIC or NODEFER PREPARE clauses.

ONCE
Specifies that DB2 determine the access path for any dynamic SQL statements only once, at the first time the statement is opened. This access path is used until the prepared statement is invalidated or removed from the dynamic statement cache and need to be prepared again.

VALIDATE RUN or VALIDATE BIND
Specifies whether to recheck, at run time, errors of the type "OBJECT NOT FOUND" and "NOT AUTHORIZED" that are found during bind or rebind. The option has no effect if all objects and needed privileges exist.

VALIDATE RUN
Specifies that if needed objects or privileges do not exist when the CREATE statement is processed, warning messages are returned, but the CREATE statement succeeds. The DB2 subsystem rechecks for the objects and privileges at run time for those SQL statements that failed the checks during processing of the CREATE statement. The authorization checks the use of the authorization ID of the owner of the routine.

VALIDATE RUN is the default.

VALIDATE BIND
Specifies that if needed objects or privileges do not exist at the time the CREATE statement is processed, an error is issued and the CREATE statement fails.

ROUNDING
Specifies the rounding mode for manipulation of DECFLOAT data. The default value is taken from the DEFAULT DECIMAL FLOATING POINT ROUNDING MODE in DECP.

DEC_ROUND_CEILING
Specifies numbers are rounded towards positive infinity.

DEC_ROUND_DOWN
Specifies numbers are rounded towards 0 (truncation).

DEC_ROUND_FLOOR
Specifies numbers are rounded towards negative infinity.

DEC_ROUND_HALF_DOWN
Specifies numbers are rounded to nearest; if equidistant, round down.

DEC_ROUND_HALF_EVEN
Specifies numbers are rounded to nearest; if equidistant, round so that the final digit is even.

DEC_ROUND_HALF_UP
Specifies numbers are rounded to nearest; if equidistant, round up.

DEC_ROUND_UP
Specifies numbers are rounded away from 0.

DATE FORMAT ISO, EUR, USA, JIS, or LOCAL
Specifies the date format for result values that are string representations of date or time values. See “String representations of datetime values” on page 101 for more information.

The default format is specified in the DATE FORMAT field of installation panel DSNTIP4 of the system where the routine is defined. You cannot use the LOCAL option unless you have a date exit routine.

DECIMAL(15), DECIMAL(31), DECIMAL(15,*s*), or DECIMAL(31,*s*)
Specifies the maximum precision that is to be used for decimal arithmetic operations. See "Arithmetic with two decimal operands" on page 244 for more information. The default format is specified in the DECIMAL ARITHMETIC field of installation panel DSNTIPF of the system where the routine is defined. If the form *pp.s* is specified, *s* must be a number between 1 and 9. *s* represents the minimum scale that is to be used for division.

FOR UPDATE CLAUSE OPTIONAL or FOR UPDATE CLAUSE REQUIRED
Specifies whether the FOR UPDATE clause is required for a DECLARE CURSOR statement if the cursor is to be used to perform positioned updates.

FOR UPDATE CLAUSE REQUIRED
Specifies that a FOR UPDATE clause must be specified as part of the cursor definition if the cursor will be used to make positioned updates.

FOR UPDATE CLAUSE REQUIRED is the default.

FOR UPDATE CLAUSE OPTIONAL
Specifies that the FOR UPDATE clause does not need to be specified in order for a cursor to be used for positioned updates. The routine body can include positioned UPDATE statements that update columns that the user is authorized to update.

The FOR UPDATE clause with no column list applies to static or dynamic SQL statements. Even if you do not use this clause, you can specify FOR UPDATE OF with a column list to restrict updates to only the columns that are named in the FOR UPDATE clause and to specify the acquisition of update locks.

TIME FORMAT ISO, EUR, USA, JIS, or LOCAL
Specifies the time format for result values that are string representations of date or time values. See "String representations of datetime values" on page 101 for more information.

The default format is specified in the TIME FORMAT field of installation panel DSNTIP4 of the system where the routine is defined. You cannot use the LOCAL option unless you have a date exit routine.

NOT SECURED or SECURED
Specifies if the function is considered secure for row access control and column access control. The SECURED or NOT SECURED option applies to all future versions of the function.

NOT SECURED
Specifies that the function is not considered secure for row access control and column access control.

NOT SECURED is the default.

When the function is invoked, the arguments of the function must not reference a column for which a column mask is enabled when the table is using active column access control.

SECURED
Specifies that the function is considered secure for row access control and column access control.

The function must be secure when it is referenced in a row permission or a column mask.

BUSINESS_TIME SENSITIVE
Determines whether references to application-period temporal tables in both

static and dynamic SQL statements are affected by the value of the CURRENT TEMPORAL BUSINESS_TIME special register.

YES
References to application-period temporal tables are affected by the value of the CURRENT TEMPORAL BUSINESS_TIME special register. YES is the default value.

NO References to application-period temporal tables are not affected by the value of the CURRENT TEMPORAL BUSINESS_TIME special register.

Related information:

"CURRENT TEMPORAL BUSINESS_TIME" on page 194

SYSTEM_TIME SENSITIVE
Determines whether references to system-period temporal tables in both static and dynamic SQL statements are affected by the value of the CURRENT TEMPORAL SYSTEM_TIME special register.

YES
References to system-period temporal tables are affected by the value of the CURRENT TEMPORAL SYSTEM_TIME special register. YES is the default value.

NO References to system-period temporal tables are not affected by the value of the CURRENT TEMPORAL SYSTEM_TIME special register.

Related information:

"CURRENT TEMPORAL SYSTEM_TIME" on page 196

ARCHIVE SENSITIVE
Determines whether references to archive-enabled tables in SQL statements are affected by the value of the SYSIBMADM.GET_ARCHIVE global variable.

YES
References to archive-enabled tables are affected by the value of the SYSIBMADM.GET_ARCHIVE global variable. YES is the default value.

NO References to archive-enabled tables are not affected by the value of the SYSIBMADM.GET_ARCHIVE global variable.

Related information:

"References to built-in global variables" on page 223

APPLCOMPAT *compatibility-level*
Specifies the package compatibility level behavior for static SQL, If this option is not specified then the behavior is determined, in priority order, by the *compatibility-level* of the last BIND or REBIND of the package or the APPLCOMPAT system parameter. The following values of *compatibility-level* can be specified:

V10R1
The static SQL statements in the package have V10R1 compatibility behavior.

V11R1
The static SQL statements in the package have V11R1 compatibility behavior.

Related information:

APPL COMPAT LEVEL field (APPLCOMPAT subsystem parameter) (DB2 Installation and Migration)

SQL-routine-body
Specifies a single SQL statement, including a compound statement. See Chapter 6, "SQL control statements for SQL routines," on page 1963 for more information about defining SQL functions.

An error is issued if an SQL function calls a procedure and the procedure issues a COMMIT, ROLLBACK, CONNECT, RELEASE, or SET CONNECTION statement.

If the *SQL-routine-body* is a compound statement, it must contain at least one RETURN statement and a RETURN statement must be executed when the function is invoked.

SQL-routine-body must not contain a period specification or period clause.

An ALTER FUNCTION (SQL scalar) statement or an ALTER PROCEDURE (SQL native) statement with an ADD VERSION clause or a REPLACE clause is not allowed in an *SQL-routine-body*.

Notes

Choosing data types for parameters:
When you choose the data types of the input and output parameters for your function, consider the rules of promotion that can affect the values of the parameters. (See "Promotion of data types" on page 110). For example, a constant that is one of the input arguments to the function might have a built-in data type that is different from the data type that the function expects, and more significantly, might not be promotable to that expected data type. Based on the rules of promotion, consider using the following data types for parameters:

- INTEGER instead of SMALLINT
- DOUBLE instead of REAL
- VARCHAR instead of CHAR
- VARGRAPHIC instead of GRAPHIC
- VARBINARY instead of BINARY

For portability of functions across platforms that are not DB2 for z/OS, do not use the following data types, which might have different representations on different platforms:

- FLOAT. Use DOUBLE or REAL instead.
- NUMERIC. Use DECIMAL instead.

Specifying the encoding scheme for parameters:
The implicitly or explicitly specified encoding scheme of all of the parameters with a character or graphic string data type (both input and output parameters) must be the same—either all ASCII, all EBCDIC, or all UNICODE.

Identifier resolution:
See Chapter 6, "SQL control statements for SQL routines," on page 1963 for information on how names are resolved to columns, SQL variables, or SQL parameters within an SQL routine.

If duplicate names are used for columns and SQL variables and parameters, qualify the duplicate names by using the table designator for columns, the routine name for parameters, and the label name for SQL variables.

Determining the uniqueness of functions in a schema:

At the current server, the function signature of each function, which is the qualified function name combined with the number and data types of the input parameters, must be unique. If the function has more than 30 input parameters, only the data types of the first 30 are used to determine uniqueness. This means that two different schemas can each contain a function with the same name that have the same data types for all of their corresponding data types. However, a single schema must not contain multiple functions with the same name that have the same data types for all of their corresponding data types.

When determining whether corresponding data types match, DB2 does not consider any length, precision, or scale attributes in the comparison. DB2 considers the synonyms of data types as a match. For example, REAL and FLOAT, and DOUBLE and FLOAT are considered a match. Therefore, CHAR(8) and CHAR(35) are considered to be the same, as are DECIMAL(11,2), DECIMAL(4,3), DECFLOAT(16) and DECFLOAT(34), TIMESTAMP(6) and TIMESTAMP(9), TIMESTAMP(6) WITH TIME ZONE and TIMESTAMP(9) WITH TIME ZONE. Furthermore, the character and graphic types, and the timestamp types are considered to be the same. For example, the following are considered to be the same type: CHAR and GRAPHIC, VARCHAR and VARGRAPHIC, CLOB and DBCLOB, TIMESTAMP WITHOUT TIME ZONE and TIMESTAMP WITH TIME ZONE. CHAR(13) and GRAPHIC(8) are considered to be the same type. An error is returned if the signature of the function being created is a duplicate of a signature for an existing user-defined function with the same name and schema.

Assume that the following statements are executed to create four functions in the same schema. The second and fourth statements fail because they create functions that are duplicates of the functions that the first and third statements created.

```
CREATE FUNCTION PART (INT, CHAR(15)) ...
CREATE FUNCTION PART (INTEGER, CHAR(40)) ...
CREATE FUNCTION ANGLE (DECIMAL(12,2)) ...
CREATE FUNCTION ANGLE (DEC(10,7)) ...
```

Overriding a built-in function:

Giving a function the same name as a built-in function is not a recommended practice unless you are trying to change the functionality of the built-in function.

If you do intend to create a function with the same name as a built-in function, be careful to maintain the uniqueness of its function signature. If your function has the same name and data types of the corresponding parameters of the built-in function but implements different logic, DB2 might choose the wrong function when the function is invoked with an unqualified function name. Thus, the application might fail, or perhaps even worse, run successfully but provide an inappropriate result.

Self-referencing function:

The body of an SQL function (that is, the *expression* or NULL in the RETURN clause of the CREATE FUNCTION statement) cannot contain a

recursive invocation of itself or to another function that invokes it, because such a function would not exist to be referenced.

Scrollable cursors specified with user-defined functions:
A row can be fetched more than once with a scrollable cursor. Therefore, if a scrollable cursor is defined with a function that is not deterministic in the select list of the cursor, a row can be fetched multiple times with different results for each fetch. Similarly, if a scrollable cursor is defined with a user-defined function with external action, the action is executed with every fetch.

Versions of a function:
The CREATE FUNCTION statement for an SQL function defines the initial version of the function. You can define additional versions using the ADD VERSION clause of the ALTER FUNCTION statement. All versions of a function share the same function signature and the same specific name. However, the parameters names can differ between versions of a functions. Only one version of the function can be considered to be the active version of the function.

Considerations for packages:
A package is generated for most SQL scalar functions. An exception is when the body of the function contains only *simple-return-statement*, and the options are limited to the following clauses:

- LANGUAGE SQL
- SPECIFIC
- PARAMETER CCSID
- NOT DETERMINISTIC
- DETERMINISTIC
- EXTERNAL ACTION
- NO EXTERNAL ACTION
- READS SQL DATA
- CONTAINS SQL
- STATIC DISPATCH
- CALLED ON NULL INPUT
- SECURED
- NOT SECURED

When the function conforms to the previous conditions, the function is created as an inline function and a package is not created.

A package is created for a non-inline function. The package that is associated with the first version of a function is named as follows:

- *location* is set to the value of the CURRENT SERVER special register.
- *collection-id* (schema) for the package is the same as the schema qualifier of the function.
- *package-id* is the same as the specific name of the function.
- *version-id* is the same as the version identifier for the initial version of the function.

The package is generated using the bind options that correspond to the implicitly or explicitly specified function options. In addition to the corresponding bind options, the package is generated using the following bind options:

- FLAG(I)
- SQLERROR(NOPACKAGE)
- ENABLE(*)

Correspondence of function options to bind command options:

The following table lists options for CREATE FUNCTION and ALTER FUNCTION and the corresponding bind command option. See BIND and REBIND options (DB2 Commands) for information about the BIND command options.

Table 110. Correspondence of function options to bind options

CREATE FUNCTION or ALTER FUNCTION option	**bind command option**
APPLICATION ENCODING SCHEME	ENCODING(ASCII), ENCODING(EBCDIC), ENCODING(UNICODE)
ARCHIVE SENSITIVE NO	ARCHIVESENSITIVE(NO)
ARCHIVE SENSITIVE YES	ARCHIVESENSITIVE(YES)
BUSINESS_TIME SENSITIVE NO	BUSTIMESENSITIVE(NO)
BUSINESS_TIME SENSITIVE YES	BUSTIMESENSITIVE(YES)
CURRENTDATA NO	CURRENTDATA(NO)
CURRENTDATA YES	CURRENTDATA(YES)
DYNAMICRULES	DYNAMICRULES(RUN), DYNAMICRULES(BIND), DYNAMICRULES(DEFINEBIND), DYNAMICRULES(DEFINERUN), DYNAMICRULES(INVOKEBIND), DYNAMICRULES(INVOKERUN)
ISOLATION LEVEL	ISOLATION(RR), ISOLATION(RS), ISOLATION(CS), ISOLATION(UR)
OPTHINT	OPTHINT
PACKAGE OWNER	OWNER
QUALIFIER	QUALIFIER
REOPT ALWAYS	REOPT(ALWAYS)
REOPT NONE	REOPT(NONE)
REOPT ONCE	REOPT(ONCE)
ROUNDING DEC_ROUND_CEILING	ROUNDING(CEILING)
ROUNDING DEC_ROUND_DOWN	ROUNDING(DOWN)
ROUNDING DEC_ROUND_FLOOR	ROUNDING(FLOOR)
ROUNDING DEC_ROUND_HALF_DOWN	ROUNDING(HALFDOWN)
ROUNDING DEC_ROUND_HALF_EVEN	ROUNDING(HALFEVEN)
ROUNDING DEC_ROUND_HALF_UP	ROUNDING(HALFUP)
ROUNDING DEC_ROUND_UP	ROUNDING(UP)
SQL PATH	PATH
SYSTEM_TIME SENSITIVE NO	SYSTIMESENSITIVE(NO)
SYSTEM_TIME SENSITIVE YES	SYSTIMESENSITIVE(YES)
VALIDATE BIND	VALIDATE(BIND)
VALIDATE RUN	VALIDATE(RUN)

Table 110. Correspondence of function options to bind options (continued)

CREATE FUNCTION or ALTER FUNCTION option	bind command option
WITH EXPLAIN	EXPLAIN(YES)
WITHOUT EXPLAIN	EXPLAIN(NO)
WITH IMMEDIATE WRITE	IMMEDWRITE(YES)
WITHOUT IMMEDIATE WRITE	IMMEDWRITE(NO)

Considerations for SQL processor programs:

SQL processor programs, such as SPUFI, the command line processor, and DSNTEP2, might not correctly parse SQL statements in the routine body that end with semicolons. These processor programs accept multiple SQL statements as input, with each statement separated with a terminator character. Processor programs that use a semicolon as the SQL statement terminator can truncate a CREATE FUNCTION statement with embedded semicolons and pass only a portion of it to DB2. Therefore, you might need to change the SQL terminator character for these processor programs. For information on changing the terminator character for SPUFI and DSNTEP2, see Setting the SQL terminator character in a SPUFI input data set (DB2 Application programming and SQL).

Lines within definitions of an SQL function:

When a non-inline SQL function is created, information is retained on lines in the CREATE statement. Lines are determined by the presence of the new line control character.

Invoking the function:

If a function is specified in the *select-list* of a *select-statement* and is the function specifies **EXTERNAL ACTION** or **MODIFIES SQL DATA**, the function will only be invoked for each row that is returned. Otherwise, the function might be invoked for rows that are not selected.

Error handling in SQL functions:

You should consider the possible exceptions that can occur for each SQL statement in the body of a non-inline SQL function. Any exception SQLSTATE that is not handled within the function (using a handler), results in the exception SQLSTATE being returned for the SQL statement that caused the function to be invoked.

Dependent objects:

An SQL routine is dependent on objects that are referenced in the routine body.

Considerations for a function that is defined using a TABLE LIKE *name* AS LOCATOR clause:

If a function is defined with a table parameter (the **TABLE LIKE** *name* **AS LOCATOR** clause was specified in the CREATE FUNCTION statement to indicate that one of the input parameters is a transition table), the function cannot be changed with an ALTER FUNCTION statement if the change requires that the parameter list be specified. For example, to add or replace a version of an SQL scalar function, the function must be dropped and re-created.

Functions and global variables:

The content of global variables that are referenced in functions is inherited from the caller. Global variables cannot be modified in or by functions.

Considerations for a function with a parameter that is defined as an array type:
A function that is defined with a parameter or RETURNS *data-type2* that is an array type can be invoked only from within an SQL PL context.

Creating a secure function:
Typically, the security administrator will examine the data that is accessed by a function, ensure that it is secure, and grant the CREATE_SECURE_OBJECT privilege to someone who currently requires the privileges to create a secure user-defined function. After the function is created, they will revoke the CREATE_SECURE_OBJECT privilege from the function owner.

DB2 treats the SECURED attribute as an assertion that declares that the security administrator has established an audit procedure for all changes to the user-defined function. DB2 assumes that such a control audit procedure is in place for all subsequent ALTER FUNCTION statements or changes to external packages. If the function is a non-inline SQL function, DB2 assumes that such a control audit procedure is in place for all versions of the function, and that all subsequent ALTER FUNCTION statements or changes to external packages are being reviewed by this audit process.

Invoking other user-defined functions in a secure function:
When a secure user-defined function is referenced in an SQL data change statement that references a table that is using row access control or column access control, and if the secure user-defined function invokes other user-defined functions, the nested user-defined functions are not validated as secure. If those nested functions can access sensitive data, the security administrator needs to ensure that those functions are allowed to access sensitive data and should ensure that a change control audit procedure has been established for all changes to those functions.

The SECURE column in the DSN_FUNCTION_TABLE EXPLAIN table:
The SECURE column in the DSN_FUNCTION_TABLE EXPLAIN table indicates if a user-defined function is considered secure.

Deploying a non-inline SQL scalar function:
When a BIND DEPLOY command is issued to deploy a non-inline SQL scalar function to a target location, the SECURED and NOT SECURED options are included in the deployment process.

When deploying a non-inline SQL scalar function, if a function with the same target name does not exist at the target location, the deployed function is created as a new function at the target location with the same SECURED or NOT SECURED option that is specified (or the default of NOT SECURED is used) in the source of the deployment.

When deploying a non-inline SQL scalar function, if a function with the same target name already exists at the target location, the deployed function is either added as a new version of the function, or the deployed function is used to replace an existing version of the function. The SECURED or NOT SECURED option of the deployed function must be the same as that of the existing function at the target location.

Alternative syntax and synonyms:
To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords:
- VARIANT as a synonym for NOT DETERMINISTIC
- NOT VARIANT as a synonym for DETERMINISTIC
- NOT NULL CALL as a synonym for RETURNS NULL ON NULL INPUT

- NULL CALL as a synonym for CALLED ON NULL INPUT
- TIMEZONE can be specified as an alternative to TIME ZONE.

For an inline SQL scalar function, the RETURNS clause and the clauses in the *option-list* can be specified in any order. For a non-inline SQL scalar function, the RETURNS clause must precede the *options-list*. For both inline and non-inline SQL scalar functions, the *RETURN-statement* must be specified after the RETURNS clause and the *options-list* in the routine body.

Examples

Example 1: Define a scalar function that returns the tangent of a value using existing SIN and COS built-in functions:

```
CREATE FUNCTION TAN (X DOUBLE)
  RETURNS DOUBLE
  LANGUAGE SQL
  CONTAINS SQL
  NO EXTERNAL ACTION
  DETERMINISTIC
  RETURN SIN(X)/COS(X);
```

Example 2: Define a scalar function that returns the text of an input string, in reverse order:

```
CREATE FUNCTION REVERSE(INSTR VARCHAR(4000))
  RETURNS VARCHAR(4000)
  DETERMINISTIC NO EXTERNAL ACTION CONTAINS SQL
  BEGIN
  DECLARE REVSTR, RESTSTR VARCHAR(4000) DEFAULT '';
  DECLARE LEN INT;
  IF INSTR IS NULL THEN
  RETURN NULL;
  END IF;
  SET (RESTSTR, LEN) = (INSTR, LENGTH(INSTR));
  WHILE LEN > 0 DO
  SET (REVSTR, RESTSTR, LEN)
    = (SUBSTR(RESTSTR, 1, 1) CONCAT REVSTR,
    SUBSTR(RESTSTR, 2, LEN - 1),
    LEN - 1);
 END WHILE;
 RETURN REVSTR;
END#
```

CREATE FUNCTION (SQL table)

The CREATE FUNCTION (SQL table) statement creates an SQL table function at the current server. The function returns a set of rows.

Invocation

This statement can only be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set that is defined below must include at least one of the following privileges or authorities:

- The CREATEIN privilege on the schema
- SYSADM authority
- SYSCTRL authority
- System DBADM

The authorization ID that matches the schema name implicitly has the CREATEIN privilege on the schema.

If the authorization ID that is used to create the function has installation SYSADM authority, the function is identified as system-defined function.

If a distinct type is referenced (i.e. as the data type of a parameter or SQL variable), the privilege set must also include at least one of the following:

- Ownership of the distinct type
- The USAGE privilege on the distinct type
- SYSADM authority

If the function uses a table as a parameter, the privilege set must also include at least one of the following:

- Ownership of the table
- The SELECT privilege on the table
- SYSADM authority

At least one of the following additional privileges is required if the SECURED option is specified

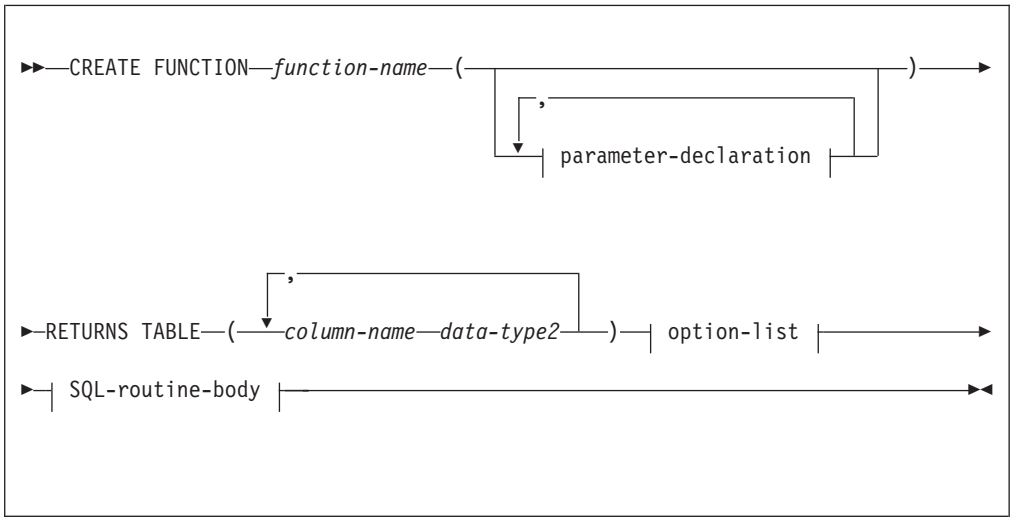
- SECADM authority
- CREATE_SECURE_OBJECT privilege

Privilege set: If the statement is embedded in an application program, the privilege set is the set of privileges that are held by the owner of the plan or package. If the owner is a role, matching of the implicit schema name does not apply and the role must include one of the previously listed privileges or authorities.

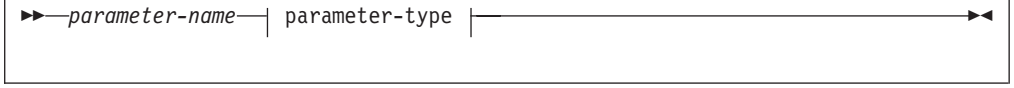
If the statement is dynamically prepared and is not running in a trusted context for which the ROLE AS OBJECT OWNER clause is specified, the privilege set is the set of privileges that are held by the SQL authorization ID of the process. If the schema name is not the same as the SQL authorization ID of the process, one of the following conditions must be met:

- The privilege set includes SYSADM authority
- The privilege set includes SYSCTRL authority
- The SQL authorization ID of the process has the CREATEIN privilege on the schema

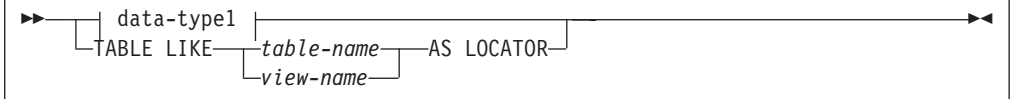
Syntax



parameter-declaration:

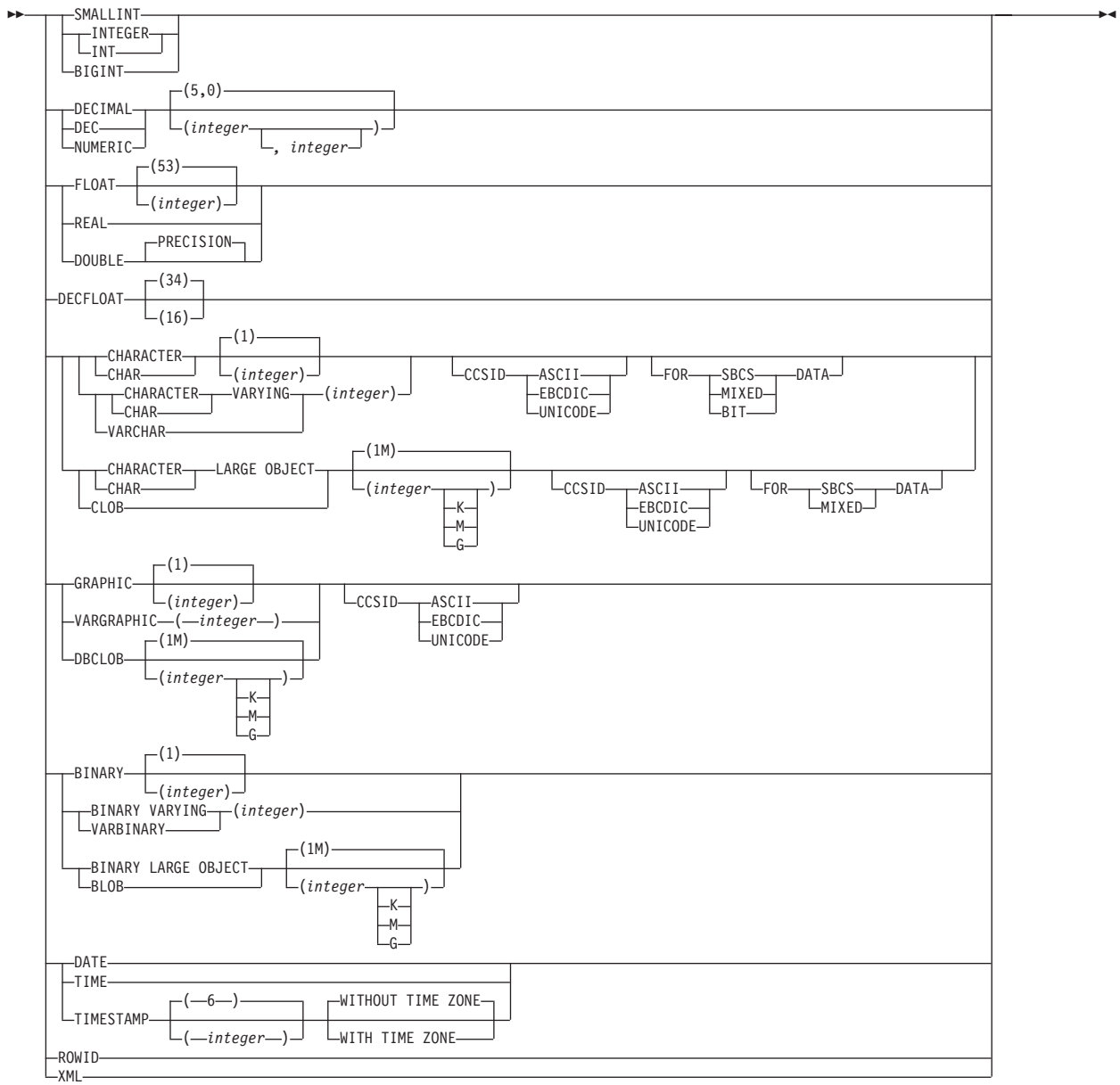


parameter-type:

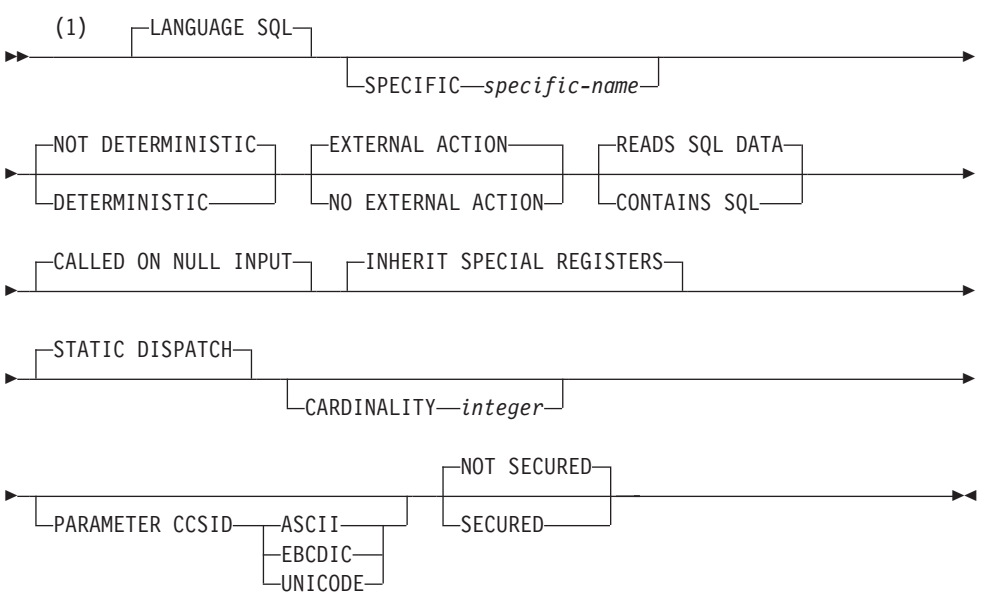


data-type1, data-type2:

built-in-type:



option-list:



Notes:

1 The options in the *option-list* can be specified in any order. However, the same clause cannot be specified more than one time.

SQL-routine-body:

Description

function-name
Names the user-defined function. The name is implicitly or explicitly qualified by a schema name. The combination of the name, the schema name, the number of parameters, and the data type of each parameter (without regard to any length, precision, scale, subtype, or encoding scheme attribute of the data type) must not identify a user-defined function that exists at the current server.

(*parameter-declaration***,...)**
Identifies the number of input parameters of the function, and specifies the name and data type of each parameter. All of the parameters for a function are input parameters and are nullable. There must be one entry in the list for each parameter that the function expects to receive.

parameter-name
Specifies the name of the input parameter. Each name in the parameter list must not be the same as any other name.

data-type1
Specifies the data type of the input parameter. The data type can be a built-in data type or a distinct type.

built-in-type
The data type of the parameter is a built-in data type.

For more information on the data types, including the subtype of character data types (the FOR subtype DATA clause), see built-in types. However, the varying length string data types have different maximum lengths for this statement than for the CREATE TABLE statement. The maximum lengths for parameters (and SQL variables) are as follows:

- VARCHAR or VARBINARY: 32704
- VARGRAPHIC: 16352

For parameters with a character or graphic data type, the PARAMETER CCSID clause or the CCSID clause indicates the encoding scheme of the parameter. If you do not specify either of the CCSID clauses, the encoding scheme is the value of the DEF ENCODING SCHEME field on installation panel DSNTIPF.

Although an input parameter with a character data type has an implicitly or explicitly specified subtype (BIT, SBCS, or MIXED), the value that is actually passed in the input parameter can have any subtype. Therefore, conversion of the input data to the subtype of the parameter might occur when the function is invoked. With ASCII or EBCDIC, an error occurs if mixed data that actually contains DBCS characters is used as the value for an input parameter that is declared with an SBCS subtype.

distinct-type-name
The data type of the parameter is a distinct type. Any length, precision, scale, subtype, or encoding scheme attributes for the parameter are those of the source type for the distinct type.

TABLE LIKE *table-name* **AS LOCATOR**
Specifies that the parameter is a transition table. However, when the function is invoked, the actual values in the transition table are not passed to the function. A single value is passed instead. This value is a locator for the table, which the function uses to access the columns of the transition table. The table that is identified can contain XML columns; however, the function cannot reference those XML columns.

A function with a table parameter can only be invoked from the triggered action of a trigger.

RETURNS TABLE
Specifies that the output of the function is a table. The RETURN statement in an SQL table function must return a table result. The parentheses that follow the RETURNS TABLE keyword delimit a list of name and data type pairs of the columns of the output table. All parameters for a function are input parameters and are nullable.

column-name
Specifies the name of the column. The name cannot be qualified, and must be unique within the RETURNS TABLE clause for the function.

data-type2
Specifies the data type and attributes of the column of the output table.

For SQL table functions, the result table of the function might include multiple encoding schemes – similar to what a view definition can include.

LANGUAGE SQL
Specifies that the function is written exclusively in SQL.

SPECIFIC *specific-name*
Specifies a unique name for the function.

NOT DETERMINISTIC or DETERMINISTIC
Specifies whether the function returns the same results each time that the function is invoked with the same input arguments. DB2 does not verify that the function program is consistent with the specification of NOT DETERMINISTIC or DETERMINISTIC.

NOT DETERMINISTIC
Specifies that the function might not return the same result table each time that the function is invoked with the same input arguments, even when the referenced data in the database has not changed. The function depends on some state values that might affect the results. DB2 uses this information to disable the merging of views and table expressions when processing SELECT and SQL data change statements that refer to this function. An example of a table function that is not deterministic is one which references special registers, other functions that are not deterministic, or a sequence in a way that affects the table function's result table. **NOT DETERMINISTIC** is the default.

DETERMINISTIC
Specifies that the function always returns the same result table each time that the function is invoked with the same input arguments (provided that the referenced data in the database has not changed). DB2 uses this information to enable the merging of views and table expressions for SELECT and SQL data change statements that refer to this function.

EXTERNAL ACTION or NO EXTERNAL ACTION
Specifies whether the function contains an external action. DB2 does not verify that the function program is consistent with the specification of EXTERNAL ACTION or NO EXTERNAL ACTION.

EXTERNAL ACTION
The function performs some external action (outside the scope of the function program). Thus, the function must be invoked with each successive function invocation. EXTERNAL ACTION must be specified if the function invokes another function that has external actions. **EXTERNAL ACTION** is the default.

NO EXTERNAL ACTION
The function does not perform any external action. It need not be called with each successive function invocation. Functions that are defined with NO EXTERNAL ACTION might perform better than functions that are defined with EXTERNAL ACTION because the function might not be invoked for each successive function invocation.

Although the scope of global variables are beyond the scope of the routine, global variables can be set in the routine body when NO EXTERNAL ACTION is specified.

READS SQL DATA or CONTAINS SQL
Specifies the classification of SQL statements that the function (any routine that is invoked from this function) can execute. DB2 verifies that the SQL statements that the function issues are consistent with this specification.

READS SQL DATA
Specifies that the function can execute statements with a data access indication of READS SQL DATA or CONTAINS SQL. The function cannot execute SQL statements that modify data.

READS SQL DATA is the default.

CONTAINS SQL
Specifies that the function can execute only SQL statements with a data access indication of CONTAINS SQL. The function cannot execute statements that read or modify data.

CALLED ON NULL INPUT
Specifies that the function is called regardless of whether any of the input arguments are null, making the function responsible for testing for null argument values. The function can return an empty table, depending on its logic.

CALLED ON NULL INPUT is the default.

INHERIT SPECIAL REGISTERS
Specifies that existing values of special registers are inherited upon entry to the function. INHERIT SPECIAL REGISTERS is the default.

STATIC DISPATCH
At function resolution time, DB2 chooses a function based on the static (or declared) types of the function parameters. **STATIC DISPATCH** is the default.

CARDINALITY *integer*
Specifies an estimate of the expected number of rows that the function returns. The number is used for optimization purposes. The value of *integer* must be between 0 and 2147483647.

If you do not specify CARDINALITY, DB2 assumes a finite value. The finite value is the same value that DB2 assumes for tables for which the RUNSTATS utility has not gathered statistics.

If a function has an infinite cardinality (the function never returns the end-of-table condition and always returns a row), a query that requires the end-of-table condition to work correctly will need to be interrupted.

PARAMETER CCSID
Specifies the encoding scheme for character and graphic string parameters is ASCII, EBCDIC, or UNICODE. The default encoding scheme is the value that is specifies in the CCSID clauses of the parameter list or RETURNS clause, or in the DEF ENCODING SCHEME field on installation panel DSNTIPF. This clause provides a convenient way to specify the encoding scheme for character and graphic string parameters. If individual CCSID clauses are specified for individual parameters in addition to this PARAMETER CCSID clause, the value specified in all of the CCSID clauses must be the same value that is specified in this clause. This clause also specifies the encoding scheme that is used for system-generated parameters of the routine such as message tokens and DBINFO.

NOT SECURED or SECURED
Specifies if the function is considered secure for row access control and column access control. The SECURED or NOT SECURED option applies to all future versions of the function.

NOT SECURED
Specifies that the function is not considered secure for row access control and column access control.

NOT SECURED is the default.

When the function is invoked, the arguments of the function must not reference a column for which a column mask is enabled when the table is using active column access control.

SECURED
Specifies that the function is considered secure for row access control and column access control.

The function must be secure when it is referenced in a row permission or a column mask.

SQL-routine-body

RETURN-statement
Specifies the return value of the function. A RETURN statement must be specified for an SQL table function.

ATOMIC
ATOMIC indicates that an unhandled exception condition within the RETURN statement causes the statement to be rolled back.

Notes

Identifier resolution:
See Chapter 6, "SQL control statements for SQL routines," on page 1963 for information on how names are resolved to columns, SQL variables, or SQL parameters within an SQL routine.

If duplicate names are used for columns and SQL variables and parameters, qualify the duplicate names by using the table designator for columns, the routine name for parameters, and the label name for SQL variables.

Referencing date and time special registers:
If an SQL function contains multiple references to any of the date or time special registers, all references return the same value. In addition, this value is the same value that is returned by the register invocation in the statement that invoked the function.

Scrollable cursors specified with user-defined functions:
A row can be fetched more than once with a scrollable cursor. Therefore, if a scrollable cursor is defined with a function that is not deterministic in the select list of the cursor, a row can be fetched multiple times with different results for each fetch. Similarly, if a scrollable cursor is defined with a user-defined function with external action, the action is executed with every fetch.

Dependent objects:
An SQL routine is dependent on objects that are referenced in the routine body.

Considerations for columns that are defined with a field procedure:
The body of an SQL table function must not reference a column that is defined with a field procedure, and the RETURNS clause of an SQL table function must not reference a column that is defined with a field procedure. An SQL table function must not be invoked with an expression that is derived from a column that is defined with a field procedure.

Creating a secure function:
Typically, the security administrator will examine the data that is accessed

by a function, ensure that it is secure, and grant the CREATE_SECURE_OBJECT privilege to someone who currently requires the privileges to create a secure user-defined function. After the function is created, they will revoke the CREATE_SECURE_OBJECT privilege from the function owner.

DB2 treats the SECURED attribute as an assertion that declares that the security administrator has established an audit procedure for all changes to the user-defined function. DB2 assumes that such a control audit procedure is in place for all subsequent ALTER FUNCTION statements or changes to external packages. If the function is a non-inline SQL function, DB2 assumes that such a control audit procedure is in place for all versions of the function, and that all subsequent ALTER FUNCTION statements or changes to external packages are being reviewed by this audit process.

Invoking other user-defined functions in a secure function:
When a secure user-defined function is referenced in an SQL data change statement that references a table that is using row access control or column access control, and if the secure user-defined function invokes other user-defined functions, the nested user-defined functions are not validated as secure. If those nested functions can access sensitive data, the security administrator needs to ensure that those functions are allowed to access sensitive data and should ensure that a change control audit procedure has been established for all changes to those functions.

The SECURE column in the DSN_FUNCTION_TABLE EXPLAIN table:
The SECURE column in the DSN_FUNCTION_TABLE EXPLAIN table indicates if a user-defined function is considered secure.

Functions and global variables:
The content of global variables that are referenced in functions is inherited from the caller. Global variables cannot be modified in or by functions.

Restrictions involving pending definition changes:
CREATE FUNCTION is not allowed if the function is an inline SQL table function that references a table that has pending definition changes.

Alternative syntax and synonyms:
To provide compatibility with previously releases of DB2 or other products int he DB2 family, DB2 supports the following keywords:

- VARIANT as a synonym for NOT DETERMINISTIC
- NOT VARIANT as a synonym for DETERMINISTIC
- NULL CALL as a synonym for CALLED ON NULL INPUT

Examples

Example 1: Define a table function, JTABLE, to return a table with 3 columns:

```
CREATE FUNCTION JTABLE (COLD_VALUE CHAR(9), T2_FLAG CHAR(1))
 RETURNS TABLE (COLA INT, COLB INT, COLC INT)
 LANGUAGE SQL
 SPECIFIC DEPTINFO
 NOT DETERMINISTIC
 READS SQL DATA
 RETURN
  SELECT A.COLA, B.COLB, B.COLC
   FROM TABLE1 AS A
    LEFT OUTER JOIN
    TABLE2 AS B
   ON A.COL1 = B.COL1 AND T2_FLAG = 'Y'
  WHERE A.COLD = COLD_VALUE;
```

Example 2: Define a table function that returns the employees in a specified department number. The function simply returns the employees for the requested department:

```
CREATE FUNCTION DEPTEMPLOYEES (DEPTNO CHAR(3))
 RETURNS TABLE (EMPNO CHAR(6), LASTNAME VARCHAR(15), FIRSTNAME VARCHAR(12))
 LANGUAGE SQL
 READS SQL DATA
 NO EXTERNAL ACTION
 DETERMINISTIC
 RETURN
  SELECT EMPNO, LASTNAME, FIRSTNME
   FROM YEMP
  WHERE YEMP.WORKDEPT = DEPTEMPLOYEES.DEPTNO;
```

CREATE GLOBAL TEMPORARY TABLE

The CREATE GLOBAL TEMPORARY TABLE statement creates a description of a temporary table at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privilege set that is defined below must include at least one of the following:

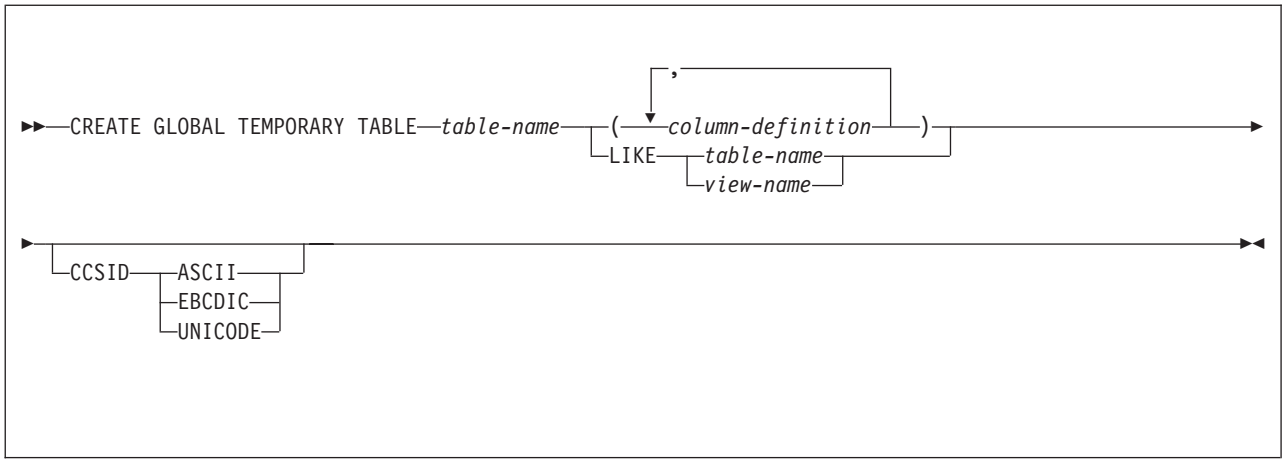
- The CREATETMTAB system privilege
- The CREATETAB database privilege for any database
- DBADM, DBCTRL, or DBMAINT authority for any database
- SYSADM or SYSCTRL authority
- System DBADM

However, DBADM, DBCTRL, or DBMAINT authority is not sufficient authority if you are creating a temporary table for someone else and the table qualifier is not your authorization ID.

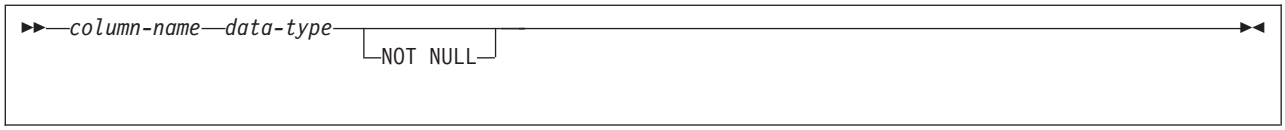
Additional privileges might be required when the data type of a column is a distinct type or the LIKE clause is specified. See the description of *distinct-type* and LIKE for the details.

Privilege set: The privilege set is the same as the privilege set for the CREATE TABLE statement. See Privilege Set for details.

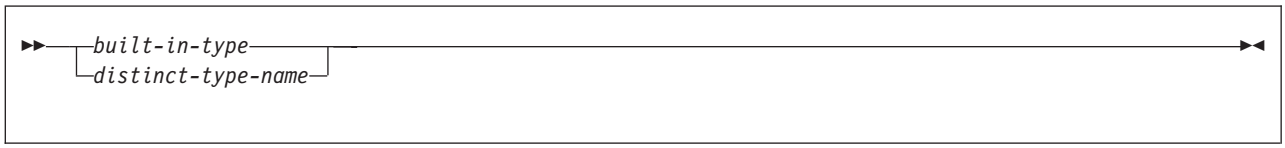
Syntax



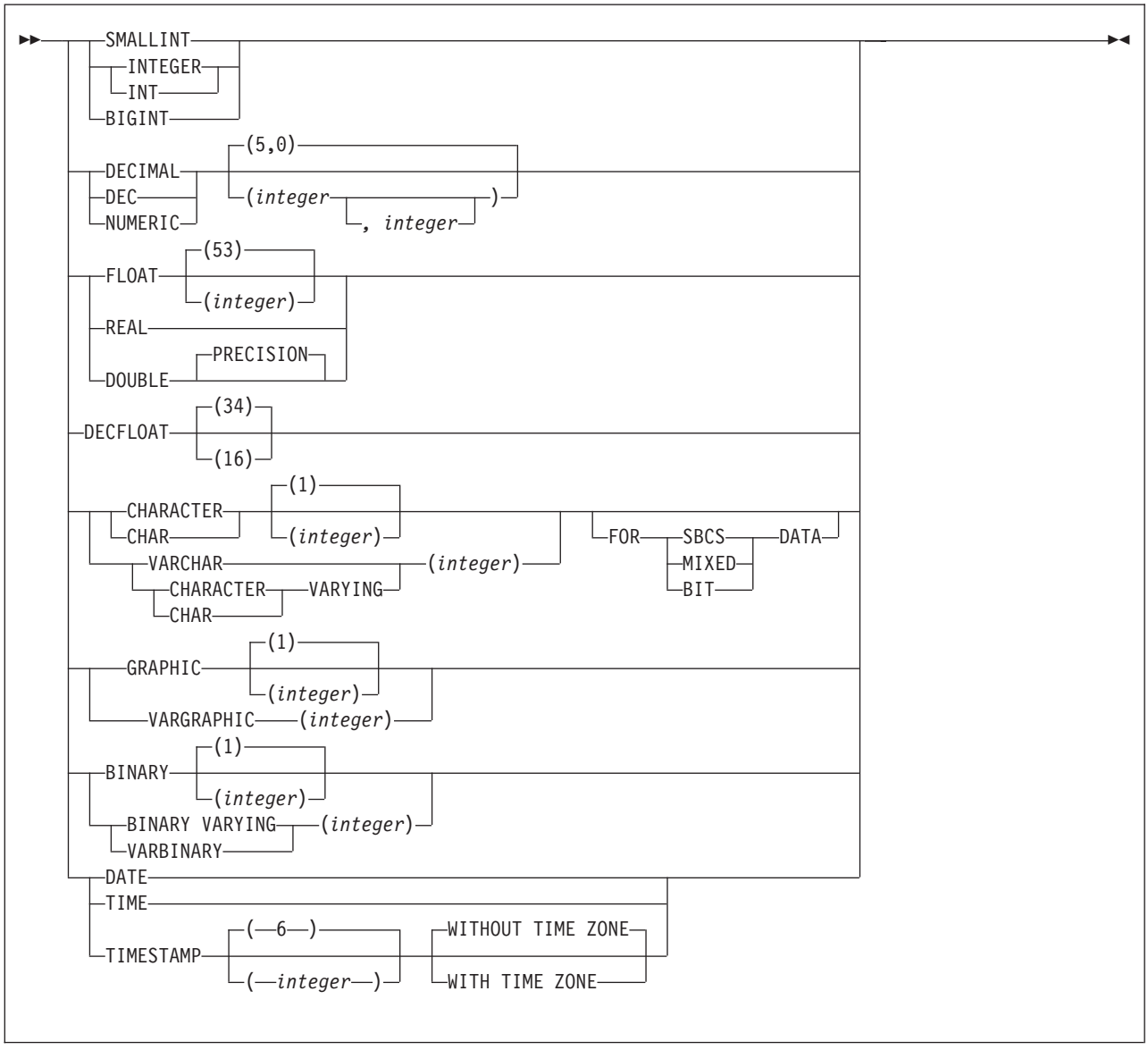
column-definition:



data-type:



built-in-type:



Description

table-name
Names the temporary table. The name, including the implicit or explicit qualifier, must not identify a table, view, alias, synonym, or temporary table that exists at the database server, or a table that exists in the SYSIBM.SYSPENDINGOBJECTS catalog table.

The qualification rules for a temporary table are the same as for other tables.

The owner acquires ALL PRIVILEGES on the table WITH GRANT OPTION and the authority to drop the table.

column-definition
Defines the attributes of a column for each instance of the table. The number of columns defined must not exceed 750. The maximum record size must not exceed 32714 bytes. The maximum row size must not exceed 32706 bytes (8 bytes less than the maximum record size).

column-name
Names the column. The name must not be qualified and must not be the same as the name of another column in the table.

data-type
Specifies the data type of the column. The data type can be a built-in data type or a distinct type.

built-in-type
The data type of the column is a built-in data type.

For more information on and the rules that apply to the data types, see built-in-type.

distinct-type
Any distinct type except one that is based on a LOB or ROWID data type. The privilege set must implicitly or explicitly include the USAGE privilege on the distinct type.

NOT NULL
Specifies that the column cannot contain nulls. Omission of NOT NULL indicates that the column can contain nulls.

LIKE *table-name* **or** *view-name*
Specifies that the columns of the table have exactly the same name and description as the columns of the identified table or view. The name specified after LIKE must identify a table, view, or temporary table that exists at the current server. A view cannot contain columns of length 0.

table-name or *view-name* must not contain a Unicode column in an EBCDIC table

The privilege set must implicitly or explicitly include the SELECT privilege on the identified table or view.

This clause is similar to the LIKE clause on CREATE TABLE, but it has the following differences:

- If any column of the identified table or view has an attribute value that is not allowed for a column in a temporary table, that attribute value is ignored. The corresponding column in the new temporary table has the default value for that attribute unless otherwise indicated.
- If any column of the identified table or view allows a default value other than null, that default value is ignored and the corresponding column in the

new temporary table has no default value. A default value other than null is not allowed for any column in a temporary table.

CCSID *encoding-scheme*
Specifies the encoding scheme for string data stored in the table.

ASCII Specifies that the data must be encoded by using the ASCII CCSIDs of the server.

An error occurs if a valid ASCII CCSID has not been specified for the installation.

EBCDIC
Specifies that data must be encoded by using the EBCDIC CCSIDs of the server.

An error occurs if a valid EBCDIC CCSID has not been specified for the installation.

UNICODE
Specifies that data must be encoded by using the CCSIDs of the server for Unicode.

An error occurs if a valid CCSID for Unicode has not been specified for the installation.

Usually, each encoding scheme requires only a single CCSID. Additional CCSIDs are needed when mixed, graphic, or Unicode data is used. An error occurs if CCSIDs have not been defined.

For the creation of temporary tables, the CCSID clause can be specified whether or not the LIKE clause is specified. If the CCSID clause is specified, the encoding scheme of the new table is the scheme that is specified in the CCSID clause. If the CCSID clause is not specified, the encoding scheme of the new table is the same as the scheme for the table specified in the LIKE clause.

Notes

Owner privileges: The owner of the table has all table privileges (see "GRANT (table or view privileges)" on page 1721) with the ability to grant these privileges to others. For more information about ownership of the object, see "Authorization, privileges, permissions, masks, and object ownership" on page 70.

Instantiation and termination: Let T be a temporary table defined at the current server and let P denote an application process:

- An empty instance of T is created as a result of the first implicit or explicit reference to T in an OPEN, SELECT INTO or SQL data change operation that is executed by any program in P.
- Any program in P can reference T and any reference to T by a program in P is a reference to that instance of T.

  When a commit operation terminates a unit of work in P and no program in P has an open WITH HOLD cursor that is dependent on T, the commit includes the operation DELETE FROM T.
- When a rollback operation terminates a unit of work in P, the rollback includes the operation DELETE FROM T.
- When the connection to the database server at which an instance of T was created terminates, the instance of T is destroyed. However, the definition of T remains. A DROP TABLE statement must be executed to drop the definition of T.

Restrictions and extensions: Let T denote a temporary table:

- Columns of T cannot have default values other than null.
- A column of T cannot have a LOB or ROWID data type (or a distinct type based on one).
- T cannot have unique constraints, referential constraints, or check constraints.
- T cannot be defined as the parent in a referential constraint.
- T cannot be referenced in:
  - A CREATE INDEX statement.
  - A LOCK TABLE statement.
  - As the object of an UPDATE statement in which the object is T or a view of T. However, you can reference T in the WHERE clause of an UPDATE statement (including the update operation of the MERGE statement).
  - DB2 utility commands.
- If T is referenced in the *fullselect* of a CREATE VIEW statement, you cannot specify a WITH CHECK OPTION clause in the CREATE VIEW statement.
- ALTER TABLE T is valid only if the statement is used to add a column to T. Any column that you add to T must have a default value of null.

  When you alter T, any packages that refer to the table are invalidated, and DB2 automatically rebinds the packages the next time they are run.
- DELETE FROM T or a *view of T* is valid only if the statement does not include a WHERE or WHERE CURRENT OF clause. In addition, DELETE FROM *view of T* is valid only if the view was created (CREATE VIEW) without the WHERE clause. A DELETE FROM statement deletes all the rows from the table or view.
- You can refer to T in the FROM clause of any subselect. If you refer to T in the first FROM clause of a *select-statement*, you cannot specify a FOR UPDATE clause.
- You cannot use a DROP DATABASE statement to implicitly drop T. To drop T, reference T in a DROP TABLE statement.
- A temporary table instantiated by an SQL statement using a three-part table name can be accessed by another SQL statement using the same name in the same application process for as long as the DB2 connection which established the instantiation is not terminated.
- GRANT ALL PRIVILEGES ON T is valid, but you cannot grant specific privileges on T.

  Of the ALL privileges, only the ALTER, INSERT, DELETE, and SELECT privileges can actually be used on T.
- REVOKE ALL PRIVILEGES ON T is valid, but you cannot revoke specific privileges from T.
- A COMMIT operation deletes all rows of every temporary table of the application process, but the rows of T are not deleted if any program in the application process has an open WITH HOLD cursor that is dependent on T. In addition, if RELEASE(COMMIT) is in effect and no open WITH HOLD cursors are dependent on T, all logical work files for T are also deleted.
- A ROLLBACK operation deletes all rows and all logical work files of every temporary table of the application process.
- You can reuse threads when using a temporary table, and a logical work file for a temporary table name remains available until deallocation. A new logical work file is not allocated for that temporary table name when the thread is reused.
- You can refer to T in the following statements:

ALTER FUNCTION	CREATE PROCEDURE	DECLARE TABLE
ALTER PROCEDURE	CREATE SYNONYM	DELETE (if it does not include a WHERE clause)
COMMENT	CREATE TABLE LIKE	DROP TABLE
CREATE ALIAS	CREATE VIEW	INSERT
CREATE FUNCTION	DESCRIBE TABLE	LABEL
		SELECT INTO

Alternative syntax and synonyms: For compatibility with previous releases of DB2, you can specify LONG VARCHAR as a synonym for VARCHAR(*integer*) and LONG VARGRAPHIC as a synonym for VARGRAPHIC(*integer*) when defining the data type of a column. However, the use of these synonyms is not encouraged because after the statement is processed, DB2 considers a LONG VARCHAR column to be VARCHAR and a LONG VARGRAPHIC column to be VARGRAPHIC.

Examples

Example 1: Create a temporary table, CURRENTMAP. Name two columns, CODE and MEANING, both of which cannot contain nulls. CODE contains numeric data and MEANING has character data. Assuming a value of NO for the field MIXED DATA on installation panel DSNTIPF, column MEANING has a subtype of SBCS:

```
CREATE GLOBAL TEMPORARY TABLE CURRENTMAP
     (CODE INTEGER NOT NULL, MEANING VARCHAR(254) NOT NULL);
```

Example 2: Create a temporary table, EMP:

```
CREATE GLOBAL TEMPORARY TABLE EMP
     (TMPDEPTNO   CHAR(3)     NOT NULL,
      TMPDEPTNAME VARCHAR(36) NOT NULL,
      TMPMGRNO    CHAR(6)             ,
      TMPLOCATION CHAR(16)            );
```

CREATE INDEX

The CREATE INDEX statement creates a partitioning index or a secondary index and an index space at the current server. The columns included in the key of the index are columns of a table at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set that is defined below must include at least one of the following:

- The INDEX privilege on the table
- Ownership of the table
- DBADM authority for the database that contains the table
- SYSADM or SYSCTRL authority
- System DBADM

If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

If the index is created using an expression, the EXECUTE privilege is required on any user-defined function that is invoked in the index expression.

Additional privileges might be required, as explained in the description of the BUFFERPOOL and USING STOGROUP clauses.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the specified index name includes a qualifier that is not the same as this owner, the privilege set must include SYSADM or SYSCTRL authority, or DBADM or DBCTRL authority for the database.

If ROLE AS OBJECT OWNER is in effect, the schema qualifier must be the same as the role, unless the role has the CREATEIN privilege on the schema, SYSADM authority, or SYSCTRL authority.

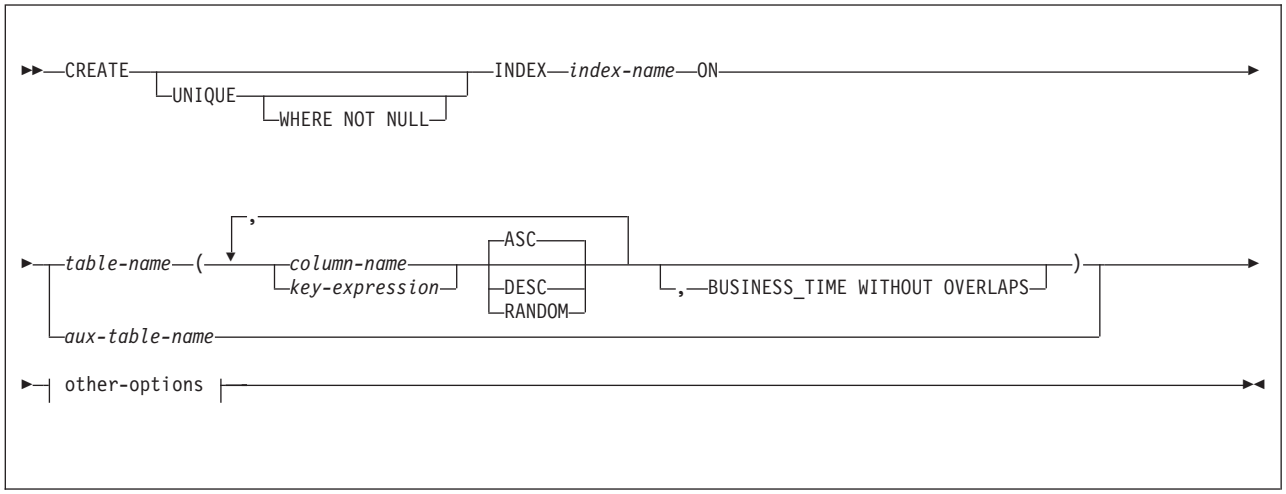
If ROLE AS OBJECT OWNER is not in effect, one of the following rules applies:

- If the privilege set lacks the CREATIN privilege on the schema, SYSADM authority, or SYSCTRL authority, the schema qualifier (implicit or explicit) must be the same as one of the authorization ids of the process.
- If the privilege set includes SYSADM authority or SYSCTRL authority, the schema qualifier can be any valid schema name.

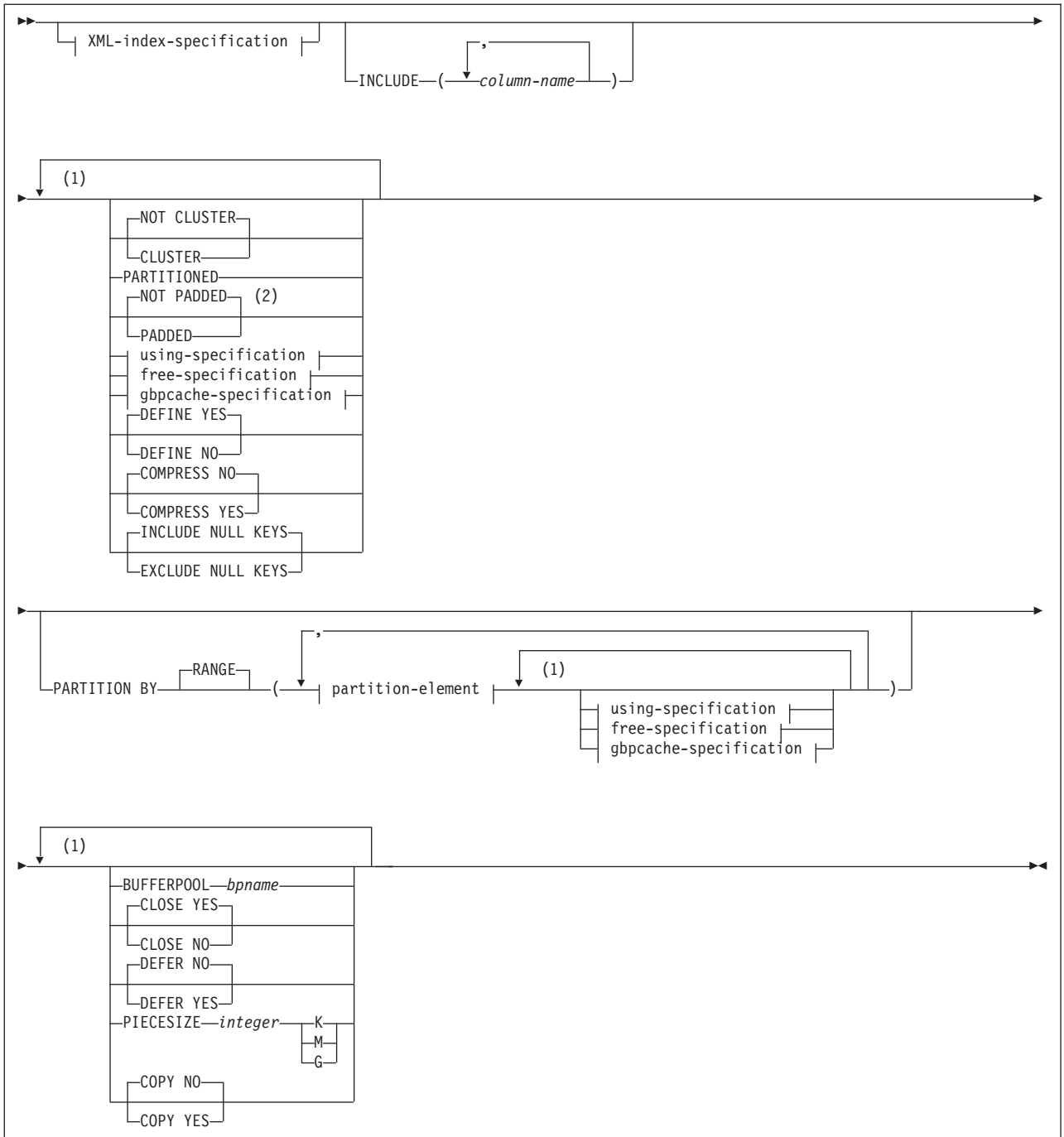
If the statement is dynamically prepared, the privilege set is the privileges that are held by the SQL authorization ID of the process unless the process is within a trusted context and the ROLE AS OBJECT OWNER clause is specified. In that case, the privilege set is the set of privileges that are held by the role that is associated with the primary authorization ID of the process. However, if the specified index name includes a qualifier that is not the same as this authorization ID, the following rules apply:

- If the privilege set includes SYSADM or SYSCTRL authority (or DBADM authority for the database, or DBCTRL authority for the database when creating a table), the schema qualifier can be any valid schema name.
- If the privilege set lacks SYSADM or SYSCTRL authority (or DBADM authority for the database, or DBCTRL authority for the database when creating a table), the schema qualifier is valid only if it is the same as one of the authorization IDs of the process and the privilege set that are held by that authorization ID includes all privileges needed to create the index. This is an exception to the rule that the privilege set is the privileges that are held by the SQL authorization ID of the process.

Syntax



other-options:

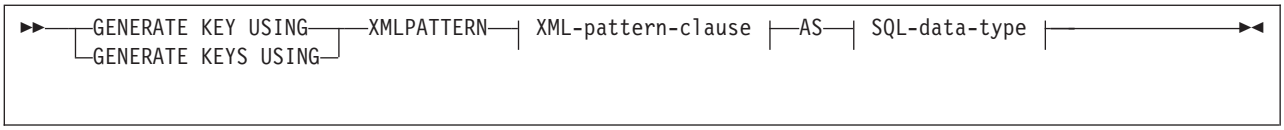


Notes:

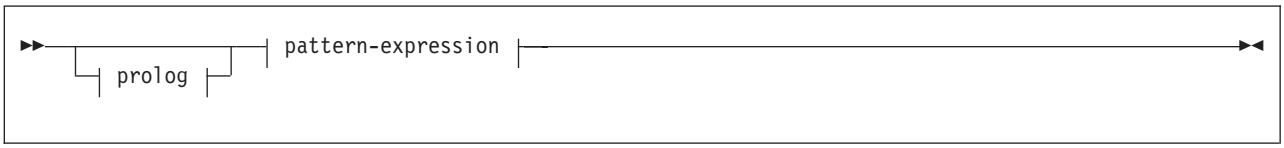
1 The same clause must not be specified more than one time.

2 The value of field PAD INDEXES BY DEFAULT (on installation panel DSNTIPE) determines the default. When the value is NO, NOT PADDED is the default. When the value is YES, PADDED is the default. For more information, see the description of the PADDED or NOT PADDED options.

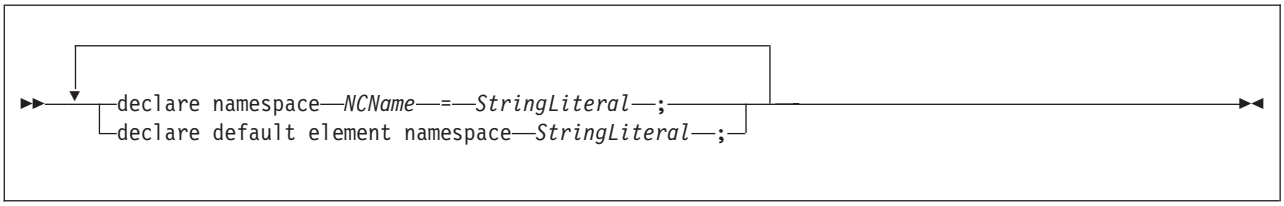
XML-index-specification:



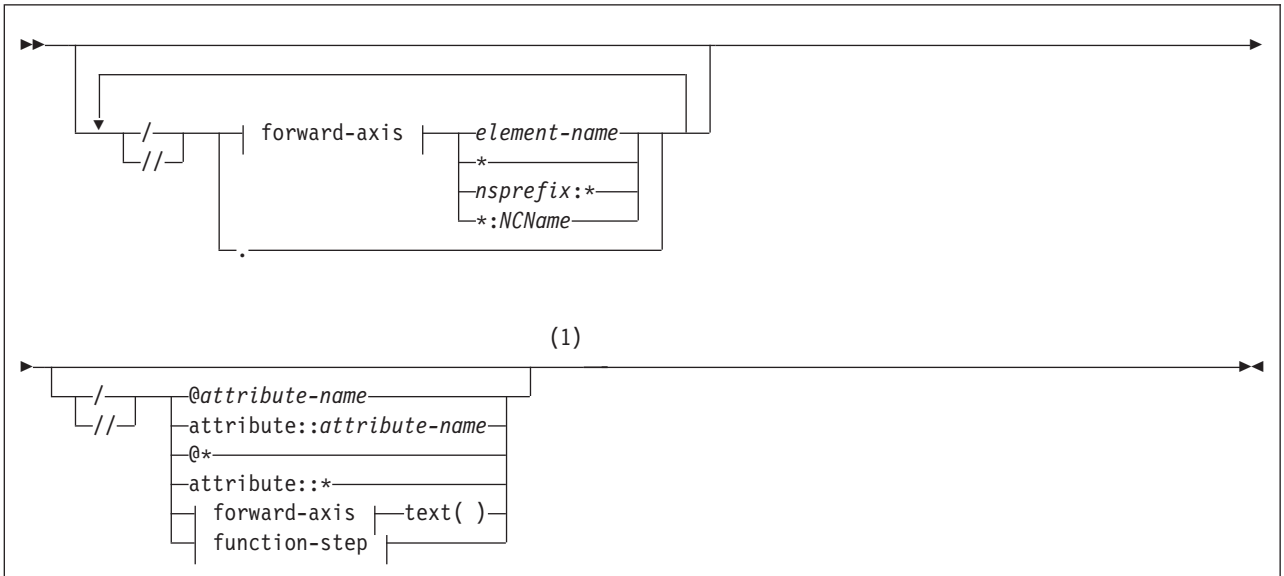
XML-pattern-clause:



prolog:



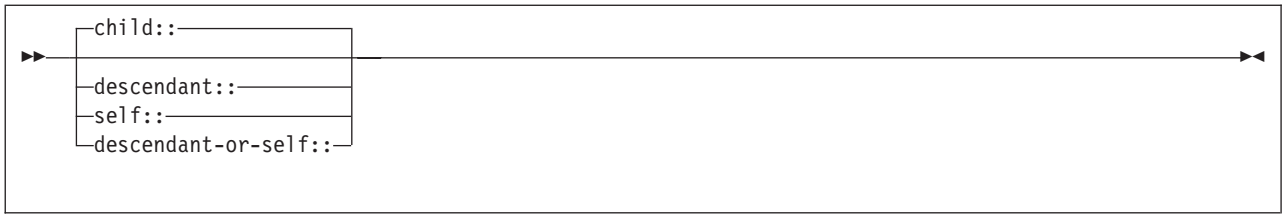
pattern-expression:



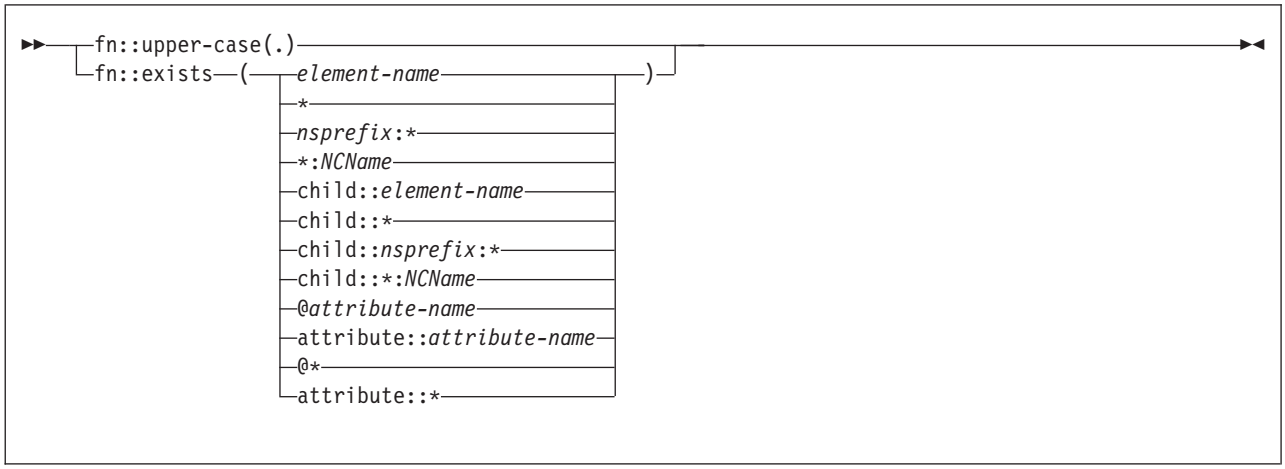
Notes:

1 *pattern-expression* cannot be an empty string.

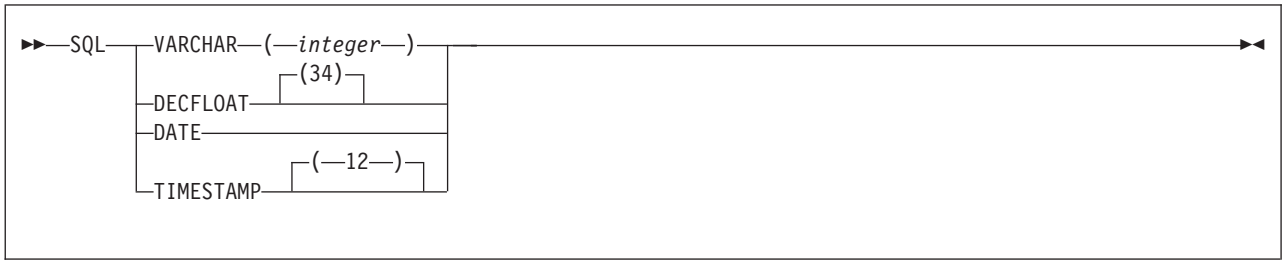
forward-axis:



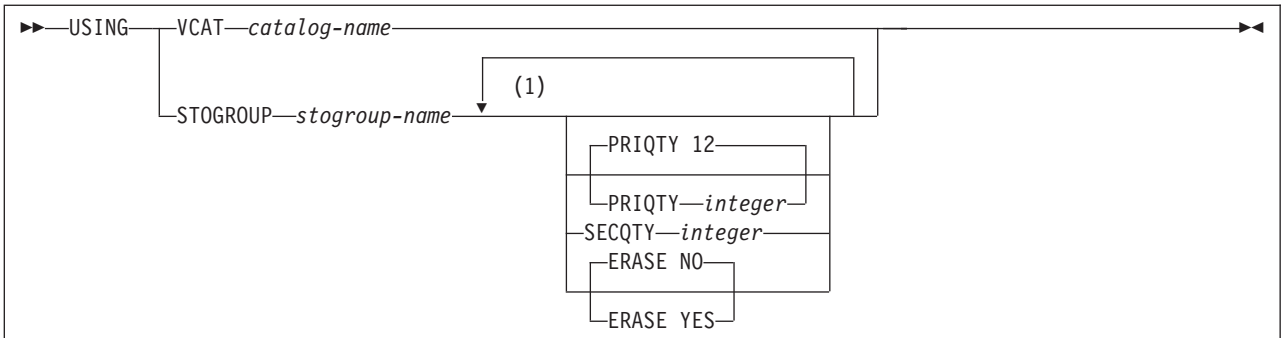
function-step:



SQL-data-type:



using-specification:



Notes:

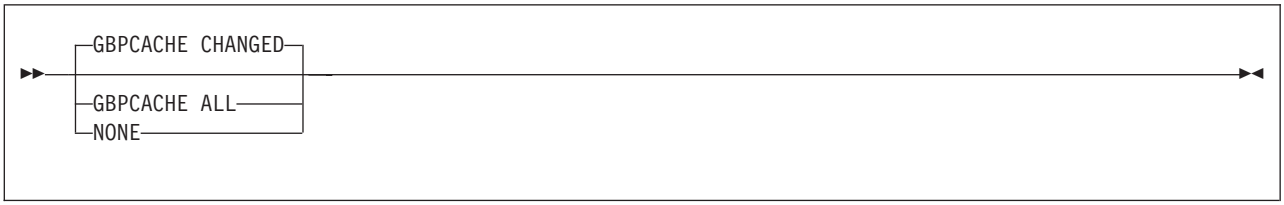
1 The same clause must not be specified more than once.

free-specification:

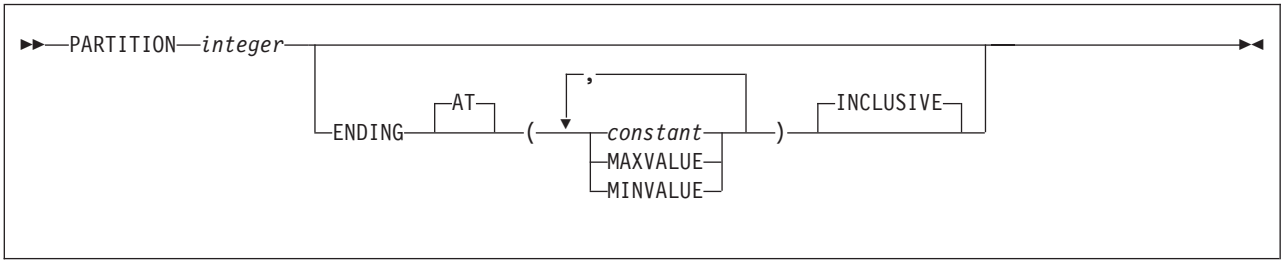
Notes:

1 The same clause must not be specified more than one time.

gbpcache-specification:



partition-element:



Description

UNIQUE
Prevents the table from containing two or more rows with the same value of the index key. When UNIQUE is used, all null values for a column are considered equal. For example, if the key is a single column that can contain null values, that column can contain only one null value. The constraint is enforced when rows of the table are updated or new rows are inserted.

The constraint is also checked during the execution of the CREATE INDEX statement. If the table already contains rows with duplicate key values, the index is not created.

UNIQUE WHERE NOT NULL
Prevents the table from containing two or more rows with the same value of the index key where all null values for a column are not considered equal. Multiple null values are allowed. Otherwise, this is identical to **UNIQUE**.

INDEX *index-name*
Names the index. The name must not identify an index that exists at the current server or in the SYSIBM.SYSPENDINGOBJECTS catalog table.

The associated index space also has a name. That name appears as a qualifier in the names of data sets defined for the index. If the data sets are managed by the user, the name is the same as the second (or only) part of *index-name*. If this identifier consists of more than eight characters, only the first eight are used. The name of the index space must be unique among the names of the index spaces and table spaces of the database for the identified table. If the data sets are defined by DB2, DB2 derives a unique name.

If the index is an index on a declared temporary table, the qualifier, if explicitly specified, must be SESSION. If the index name is unqualified, DB2 uses SESSION as the implicit qualifier.

ON *table-name* **or** *aux-table-name*
Identifies the table on which the index is created. The name can identify a base table, a materialized query table, a declared temporary table, or an auxiliary table.

table-name
Identifies the base table, materialized query table, or declared temporary table on which the index is created. The name must identify a table that exists at the current server. (The name of a declared temporary table must be qualified with SESSION.)

The name must not identify a clone table. The name must not identify a created temporary table or a table that is implicitly created for an XML column. If the index that is being created is for XML values, the table can contain an XML column, otherwise, the table must not contain an XML column. The name cannot identify a catalog table or declared temporary table if the index is created using expressions.

If the table has enforced row or column access controls, the row permissions and column masks are not applied during key generation.

column-name,...
Specifies the columns of the index key.

Each *column-name* must identify a column of the table. Do not specify more than 64 columns or the same column more than one time. Do not qualify *column-name*.

Do not specify a column for *column-name* that is defined as follows:

- a LOB column (or a column with a distinct type that is based on a LOB data type)
- a DECFLOAT column (or a column with a distinct type that is based on a DECFLOAT data type)
- a BINARY or VARBINARY column (or a column with a distinct type that is based on a BINARY or VARBINARY data type) when the PARTITION BY RANGE clause is also specified
- a VARBINARY column (or a column with a distinct type that is based on a VARBINARY data type) when the PADDED clause is also specified
- a row change timestamp column when the PARTITION BY RANGE clause is also specified.
- a timestamp with time zone column (or a column with a distinct type that is based on the timestamp with time zone data type) when the PARTITION or PARTITION BY RANGE clause is also specified.

A column with an XML type can only be specified if the XMLPATTERN clause is also specified. If the XMLPATTERN clause is specified, only one column can be identified and the column must be an XML type. The resulting index is an XML index.

If the column is a Unicode column in an EBCDIC table, the index key is converted to Unicode and the index is treated as an expression-based index, subject to the same restrictions as other expression-based indexes.

The sum of the length attributes of the columns must not be greater than the following limits, where *n* is the number of columns that can contain null values and *m* is the number of varying-length columns in the key:

- 2000 - *n* for a padded, nonpartitioning index
- 2000 - *n* - 2*m* for a nonpadded, nonpartitioning index
- 255 - *n* for a partitioning index (padded or nonpadded)
- 255 - *n* - 2*m* for a nonpadded, partitioning index

key-expression
Specifies an expression that returns a scalar value. An index with a key that includes one or more expressions consisting of more than just a column name is an *expression-based index*.*key-expression* cannot be specified with the GENERATE KEY USING clause or the INCLUDE clause. *key-expression* has the following restrictions:

- Each *key-expression* must contain as least one reference to a column of *table-name*.

  All references to columns of *table-name* must be unqualified. Referenced columns cannot be LOB, XML, or DECFLOAT data types or a distinct type that is based on one of these data types. Referenced columns cannot include any FIELDPROCs or a SECURITY LABEL. Referenced columns cannot be implicitly hidden (that is, defined with the IMPLICITLY HIDDEN attribute).
- *key-expression* must not including the following:
  - A subquery
  - An aggregate function
  - A function that is not deterministic function
  - A function that has an external action

  - A user-defined function
  - The VERIFY_GROUP_FOR_USER or VERIFY_ROLE_FOR_USER functions
  - A sequence reference
  - A host variable
  - A parameter marker
  - A special register
  - An expression for which implicit time zone value apply (or example, cast a timestamp to a timestamp with time zone)
  - A CASE expression
  - An OLAP specification
- If *key-expression* references a cast function, the privilege set must implicitly include EXECUTE authority on the generated cast functions for the distinct type.
- If *key-expression* references the LOWER or UPPER functions, the input *string-expression* cannot be FOR BIT DATA, and the function invocation must contain the *locale-name* argument.
- If *key-expression* references the TRANSLATE function, the function invocation must contain the *to-string* argument.
- If *key-expression* references the SUBSTR function, the function can reference the inline portion of a LOB column.
- The same expression cannot be used more than one time in the same index.
- The data type of the result of the expression cannot be a LOB, XML, DECFLOAT, or array value. However, the data type of the intermediate result can be a LOB value, but not an XML or DECFLOAT value.
- If a Unicode column in an EBCDIC table is referenced in a *key-expression*, the encoding scheme of the index keys must either be all Unicode or all EBCDIC. Otherwise, the encoding schema of the result of a *key-expression* must be the same encoding scheme as the table.

The maximum length of the text string of each *key-expression* is 4000 bytes after conversion to UTF-8. The maximum number of *key-expression* in an extended index is 64.

ASC
Puts the index entries in ascending order by the column. ASC cannot be specified with the GENERATE KEY USING clause.

ASC is the default.

DESC
Puts the index entries in descending order by the column. DESC cannot be specified with the GENERATE KEY USING clause or if the ON clause contains *key-expression*.

DESC cannot be specified if the column is a Unicode column in an EBCDIC table.

RANDOM
Index entries are put in a random order by the column. RANDOM cannot be specified in the following cases:

- A varying length column is part of the index key and the index is defined with the NOT PADDED option

- A column of the index key is defined as TIMESTAMP WITH TIME ZONE
- The index is an XML index. An XML index is defined with the GENERATE KEY USING clause
- The index is part of the partitioning key
- A column of the index key is a Unicode column in an EBCDIC table
- The index is an expression-based index

BUSINESS_TIME WITHOUT OVERLAPS
BUSINESS_TIME WITHOUT OVERLAPS can only be specified for an index that is defined as UNIQUE. When BUSINESS_TIME WITHOUT OVERLAPS is specified, the values for the rest of the specified keys are unique with respect to any period of time.

BUSINESS_TIME WITHOUT OVERLAPS can be specified as the last item in the list. The list must include at least one *column-name* or *key-expression*.

When BUSINESS_TIME WITHOUT OVERLAPS is specified, the columns of the BUSINESS_TIME period must not be specified as *key-expressions*.

When BUSINESS_TIME WITHOUT OVERLAPS is specified, the following columns are added to the index:

- The end column of the BUSINESS_TIME period in ascending order
- The start column of the BUSINESS_TIME period in ascending order

BUSINESS_TIME WITHOUT OVERLAPS must not be specified for a PARTITIONED index.

aux-table-name
Identifies the auxiliary table on which the index is created. The name must identify an auxiliary table that exists at the current server. If the auxiliary table already has an index, do not create another one. An auxiliary table can only have one index.

Do not specify any columns for the index key. The key value is implicitly defined as a unique 19 byte value that is system generated.

If qualified, *table-name* or *aux-table-name* can be a two-part or three-part name. If a three-part name is used, the first part must match the value of the field DB2 LOCATION NAME of installation panel DSNTIPR at the current server. (If the current server is not the local DB2, this name is not necessarily the name in the CURRENT SERVER special register.) Whether the name is two-part or three-part, the authorization ID that qualifies the name is the owner of the index.

The table space that contains the named table must be available to DB2 so that its data sets can be opened. If the table space is EA-enabled, the data sets for the index must be defined to belong to a DFSMS data class that has the extended format and addressability attributes.

GENERATE KEY USING
Along with XMLPATTERN, GENERATE KEY USING is required to generate an XML index.

XMLPATTERN
When an XML column is indexed, only parts of the documents will be indexed. To identify those parts, a path expression that follows the

XMLPATTERN clause is specified. Only values of those element, attribute, or text nodes which match the specified pattern are indexed. An XML pattern can be specified using an optional namespace declaration where namespace prefixes are mapped to namespace URIs and by providing a path expression. The path expression is similar to a path expression in XQuery except that the paths that are specified for the XML index can support child axis, self-or-descendant axis, wildcard expressions, or attribute only. The maximum length of an XML pattern text is 4000 bytes after being converted to UTF-8. Refer to *DB2 XML Guide* for more information about XQuery.

prolog
To use qualified names in the *pattern-expression*, namespace prefixes need to be declared. A default namespace can also be declared for use with unqualified names.

declare namespace *NCName=StringLiteral*
The namespace prefix, *NCName*, is mapped to a namespace URI that is identified in *StringLiteral*. Multiple namespaces can be declared, but each namespace prefix must be unique within the list of namespace declarations. *NCName* is an XML name as defined by the XML 1.0 standard. *NCName* cannot include a colon character. The namespace URI cannot be `http://www.w3.org/XML/1998/namespace` or `http://w3.org/2000/xmlns/`.

declare default element namespace *StringLiteral*
Specifies the default namespace URI for unqualified names of elements and types. *StringLiteral* is a namespace URI. If no default element namespace is declared, unqualified names of element and types are in no namespace. Only one default namespace can be declared.

pattern-expression
Pattern-expression is used to identify those nodes in an XML document that are indexed. *Pattern-expression* cannot be an empty or invalid string, and the XQuery expression cannot be nested more than 50 levels. *pattern-expression* cannot be an XQuery updating expression.

/ (forward slash)
Separates path expression steps.

// (double forward slash)
Abbreviated syntax for /descendant-or-self::node()/

. (dot)
Abbreviated syntax for /self::node()/

child::
Specifies children of the context node. child:: is the default if no forward axis is specified.

descendant::
Specifies the descendants of the context node.

self::
Specifies the current context node.

descendant-or-self::
Specifies the context node and the descendents of the context node.

element-name
Identifies an element in an XML document. *element-name* is an XML QName that can have one of the following forms:

nsprefix:*NCName*
nsprefix explicitly specifies a namespace prefix that must be declared.

NCName
An unqualified XML name that uses the default namespace.

*** (an asterisk)**
Indicates any element name. If * is prefixed by attribute:: or @, * indicates any attribute name.

nsprefix:*
Indicates any NCName within the specified namespace.

*:*NCName*
Indicates a specific XML name in any of the currently declared namespaces.

attribute:: or @
Specifies attributes of the context node.

attribute-name
Identifies an attribute in an XML document. *attribute-name* is an XML QName that can have one of the following forms:

nsprefix:*NCName*
nsprefix explicitly specifies a namespace prefix that must be declared.

NCName
An unqualified XML name that uses the default namespace.

text( )
Matches any text node.

fn:upper-case(.)
Specifies an element node or an attribute node that identifies the key value for the index for each node that is specified by the context step (the part of *pattern-expression* that is specified prior to fn:upper-case).

The context step of fn:upper-case() must specify an element node or an attribute node. The argument of fn:upper-case() must be a self step. The key values of an XML value index must be specified as the SQL data type VARCHAR. The length of the VARCHAR value can be any value that is allowed in DB2.

fn:exists()
Specifies an element node that identifies the key value for the index for each node that is specified by the context step (the part of *pattern-expression* that is specified prior to fn:exists).

The context step of fn:exists() must specify an element node. The argument of fn:exists() must be either a single step of a child element node or an attribute node. The name test part can be a wildcard character for either the namespace prefix or NCName. The key values of an XML value index for an XPath expression that ends with fn:exists() must be specified as the SQL data type VARCHAR(1). The key value will be "T" or "F". "T" implies that fn:exists() evaluates to true and "F" implies that fn:exists() evaluates to false.

AS SQL *data-type*
Specifies that indexed values are stored as an instance of the specified SQL data type. Casting to the specified data type can result in a loss of precision of

the values. For example, a loss of precision can occur when an XML integer value is cast to the SQL data type DECFLOAT. If the cast causes a loss of precision, the result will be rounded to the approximate value when it is stored in the index. The cast result cannot be outside of the range that is supported by the SQL data type. If the value cannot be cast to the specified data type, the document is still inserted into the table, but the index entry for that value is not created. No error or warning code is returned.

If the index is unique, the uniqueness is enforced on the value after it is cast to the specified type. Because rounding can occur during the cast to the SQL data type, if a value is cast to the same key value as a document that the table already contains, DB2 will return duplicate key errors at insert time, or fail to create the index.

VARCHAR (*integer*)
The length *integer* is a value in the range of 1 to 1000 bytes. If VARCHAR is specified with a length, the specified length is treated as a constraint. If documents are inserted into a table (or exist in the table at create index time) that have nodes with values that are longer than the specified length, the insert or index creation will fail.

DECFLOAT
DECFLOAT can be specified to index numeric values. For the cast to succeed, the string must be a valid XML numeric type. Otherwise the value will be ignored and no insert to the index will occur. The result of the cast cannot be outside of the range that DECFLOAT can represent. Because the XML Schema data type for numeric values allows greater precision than the SQL data type, the result might be rounded to fit into the SQL data type. The DECFLOAT values that are stored in the index are the normalized numeric values.

DATE
The SQL DATE data type values will be normalized to UTC (Coordinated Universal Time) before being stored in the index. For invalid xs:date values, the value will be ignored without being inserted into the index. The XML schema data type for DATE allows for greater precision than the SQL data type. If an out-of-range value is encountered, an error is returned.

TIMESTAMP (12)
The SQL TIMESTAMP data type values will be normalized to UTC (Coordinated Universal Time) before being stored in the index. If the value that is specified in the document does not specify the time zone, DB2 will use the implicit time zone to normalize the value to UTC. For invalid xs:dateTime values, the value will be ignored without being inserted into the index. The XML schema data type for timestamps allows for greater precision than the SQL data type. If an out-of range value is encountered, an error is returned. Only a precision of 12 fractional digits is allowed for an SQL TIMESTAMP index key.

INCLUDE (*column-name*)
Specifies additional columns to append to the set of index key columns of a unique index. Any column that is specified using **INCLUDE** *column-name* is not used to enforce uniqueness. The included columns might improve performance for some queries using index only access.

The UNIQUE clause must be specified when INCLUDE is specified. Columns that are specified in the INCLUDE clause count towards the limits for the

number of columns and the limits on the sum of the length attributes of the columns that are specified in the index. The total number of columns for the index cannot exceed 64.

column-name must be distinct from the columns that are used to enforce uniqueness and from other columns specified in the INCLUDE clause. *column-name* must be unqualified, must identify a column of the specified table, and must not be one of the existing columns of the index. *column-name* must not identify a LOB or DECFLOAT column (or a distinct type that is based on one of those types).

The INCLUDE clause cannot be specified for the following types of indexes:

- A non-unique index
- A partitioning index when index-controlled partitioning is used
- An auxiliary index
- An XML index
- An extended index
- An expression-based index

CLUSTER or NOT CLUSTER

Specifies whether the index is the clustering index for the table. This clause must not be specified for an index on an auxiliary table, or on a table that is defined to use hash organization.

CLUSTER

The index is to be used as the clustering index of the table. CLUSTER cannot be specified if XMLPATTERN or *key-expression* is specified.

NOT CLUSTER

The index is not to be used as the clustering index of the table.

PARTITIONED

Specifies that the index is data partitioned (that is, partitioned according to the partitioning scheme of the underlying data). A partitioned index can be created only on a partitioned table space, not on a partition-by-growth table space. PARTITIONED cannot be specified if XMLPATTERNor if BUSINESS_TIME WITHOUT OVERLAPS is specified. The types of partitioned indexes are partitioning and secondary.

An index is considered a partitioning index if the specified index key columns match or comprise a superset of the columns specified in the partitioning key, are in the same order, and have the same ascending or descending attributes.

If PARTITION BY was not specified when the table was created, the CREATE INDEX statement must have the ENDING AT clause specified to define a partitioning index and use index-controlled partitioning. This index is created as a partitioned index even if the PARTITIONED keyword is not specified. When a partitioning index is created, if both the PARTITIONED and ENDING AT keywords are omitted, the index will be non-partitioned. If PARTITIONED is specified, the USING specification with PRIQTY and SECQTY specifications are optional. If these space parameters are not specified, default values are used.

A secondary index is any index defined on a partitioned table space that does not meet the definition of the partitioning index. For partitioned secondary indexes (data-partitioned secondary indexes), the ENDING AT clause is not allowed because the partitioning scheme of the index is predetermined by that of the underlying data. UNIQUE and UNIQUE WHERE NOT NULL are not allowed unless the columns in the index are a superset of the partitioning

columns. If a partitioned secondary index is created on a table that uses index-controlled partitioning, the table is converted to use table-controlled partitioning.

Index-controlled partitioning cannot be used if the PREVENT_NEW_IXCTRL_PART subsystem parameter is set to YES.

Related information:

PREVENT INDEX PART CREATE field (PREVENT_NEW_IXCTRL_PART subsystem parameter) (DB2 Installation and Migration)

NOT PADDED or PADDED

Specifies how varying-length string columns are to be stored in the index. If the index contains no varying-length columns, this option is ignored, and a warning message is returned. Indexes that do not have varying-length string columns are always created as physically padded indexes.

NOT PADDED

Specifies that varying-length string columns are not to be padded to their maximum length in the index. The length information for a varying-length column is stored with the key.

NOT PADDED must be used if the definition of the index refers to a Unicode column in an EBCDIC table.

NOT PADDED is ignored and has no effect if the index is being created on an auxiliary table. Indexes on auxiliary tables are always padded.

PADDED

Specifies that varying-length string columns within the index are always padded with the default pad character to their maximum length. PADDED cannot be specified if XMLPATTERN is specified. PADDED cannot be specified for indexes that are defined on VARBINARY columns.

When the index contains at least one varying-length column, the default for the option depends on the value of field PAD INDEXES BY DEFAULT on installation panel DSNTIPE:

- When the value of this field is NO, new indexes are not padded unless PADDED is specified.
- When the value of this field is YES, new indexes are padded unless NOT PADDED is specified.

The components of the USING clause are discussed below, first for non-partitioned indexes and then for partitioned indexes.

Using clause for non-partitioned indexes

For non-partitioned indexes, the USING clause indicates whether the data sets for the index are to be managed by the user or managed by DB2. If DB2 definition is specified, the clause also gives space allocation parameters (PRIQTY and SECQTY) and an erase rule (ERASE).

If you omit USING, the data sets will be managed by DB2 on volumes listed in the default storage group of the database that is associated with the table. That default storage group must exist. With no USING clause, PRIQTY, SECQTY, and ERASE assume their default values.

VCAT *catalog-name*

Specifies that the first data set for the index is managed by the user, and that following data sets, if needed, are also managed by the user.

The data sets defined for the index are linear VSAM data sets cataloged in an integrated catalog facility catalog identified by *catalog-name*. An alias[30] must be used if *catalog-name* is longer than eight characters.

Conventions for index data set names are given in *DB2 Administration Guide*. *catalog-name* is the first qualifier for each data set name.

One or more DB2 subsystems could share integrated catalog facility catalogs with the current server. To avoid the chance of having one of those subsystems attempt to assign the same name to different data sets, select a value for *catalog-name* that is not used by the other DB2 subsystems.

Do not specify VCAT for an index on a declared temporary table or if the table space is partition-by-growth.

STOGROUP *stogroup-name*

Specifies that DB2 will define and manage the data sets for the index. Each data set will be defined on a volume listed in the identified storage group. The values specified (or the defaults) for PRIQTY and SECQTY determine the primary and secondary allocations for the data set. If PRIQTY+118×SECQTY is 2 gigabytes or greater, more than one data set could eventually be used, but only the first is defined during execution of this statement.

To use USING STOGROUP, the privilege set must include SYSADM authority, SYSCTRL authority, or the USE privilege for that storage group. Moreover, *stogroup-name* must identify a storage group that exists at the current server and includes in its description at least one volume serial number. The description can indicate that the choice of volumes will be left to Storage Management Subsystem (SMS). Each volume specified in the storage group must be accessible to z/OS for dynamic allocation of the data set, and all these volumes must be of the same device type.

The integrated catalog facility catalog used for the storage group must ***not*** contain an entry for the first data set of the index. If the catalog is password protected, the description of the storage group must include a valid password.

The storage group supplies the data set name. The first level qualifier is also the name of, or an alias for, the integrated catalog facility catalog on which the data set is to be cataloged. The naming convention for the data set is the same as if the data set is managed by the user.

Do not specify the STOGROUP option when you create indexes on the DB2 catalog. User-created indexes on the catalog should be user-managed.

PRIQTY *integer*

Specifies the minimum primary space allocation for a DB2-managed data set. When you specify PRIQTY (with a value other than -1), the primary space allocation is at least *n* kilobytes, where *n* is:

12 If *integer* is less than 12

integer

If *integer* is between 12 and 4194304

30. The alias of an integrated catalog facility catalog

2097152
If both of the following conditions are true:
- *integer* is greater than 2097152.
- The index is a non-partitioned index on a table space that is not defined with the LARGE or DSSIZE attribute.

4194304
If *integer* is greater than 4194304

If you do not specify PRIQTY or specify PRIQTY -1, DB2 uses a default value for the primary space allocation; for information on how DB2 determines the default value, see Rules for primary and secondary space allocation.

If you specify PRIQTY and do not specify a value of -1, DB2 specifies the primary space allocation to access method services using the smallest multiple of 4KB not less than *n*. The allocated space can be greater than the amount of space requested by DB2. For example, it could be the smallest number of tracks that will accommodate the space requested. To more closely estimate the actual amount of storage, see DEFINE CLUSTER command (DFSMS Access Method Services for Catalogs).

When determining a suitable value for PRIQTY, be aware that two of the pages of the primary space could be used by DB2 for purposes other than storing index entries.

SECQTY *integer*
Specifies the minimum secondary space allocation for a DB2-managed data set. If you do not specify SECQTY, DB2 uses a formula to determine a value. For information on the actual value that is used for secondary space allocation, whether you specify a value or not, see Rules for primary and secondary space allocation.

If you specify SECQTY and do not specify a value of -1, DB2 specifies the secondary space allocation to access method services using the smallest multiple of 4KB not less than *n*. The allocated space can be greater than the amount of space requested by DB2. For example, it could be the smallest number of tracks that will accommodate the space requested. To more closely estimate the actual amount of storage, see DEFINE CLUSTER command (DFSMS Access Method Services for Catalogs).

ERASE
Indicates whether the DB2-managed data sets are to be erased when they are deleted during the execution of a utility or an SQL statement that drops the index.

NO Does not erase the data sets. Operations involving data set deletion will perform better than ERASE YES. However, the data is still accessible, though not through DB2. This is the default.

YES
Erases the data sets. As a security measure, DB2 overwrites all data in the data sets with zeros before they are deleted.

USING clause for partitioned indexes:
If the index is partitioned, there is a PARTITION clause for each partition. Within a PARTITION clause, a USING clause is optional. If a USING clause is present, it applies to that partition in the same way that a USING clause for a secondary index applies to the entire index.

When a USING specification is absent from a PARTITION clause, the USING clause parameters for the partition depend on whether a USING clause is specified before the PARTITION clauses.

- If the USING clause is specified, it applies to every PARTITION clause that does not include a USING clause.
- If the USING clause is not specified, the following defaults apply to the partition:
  - Data sets are managed by DB2
  - The default storage group for the database is used
  - A value of 12 is used for PRIQTY and SECQTY
  - A value of NO is used for ERASE

VCAT *catalog-name*
Specifies a user-managed data set with a name that starts with the specified catalog name. You must specify an alias for the integrated catalog facility catalog if the name of the integrated catalog facility catalog is longer than eight characters.

If *n* is the number of the partition, the identified integrated catalog facility catalog must already contain an entry for the *n*th data set of the index, conforming to the DB2 naming convention for data sets set forth in *DB2 Administration Guide*.

One or more DB2 subsystems could share integrated catalog facility catalogs with the current server. To avoid the chance of having one of those subsystems attempt to assign the same name to different data sets, select a value for *catalog-name* that is not used by the other DB2 subsystems.

DB2 assumes one and only one data set for each partition.

STOGROUP *stogroup-name*
If USING STOGROUP is used, explicitly or by default, for a partition *n*, DB2 defines the data set for the partition during the execution of the CREATE INDEX statement, using space from the named storage group. The privilege set must include SYSADM authority, SYSCTRL authority, or the USE privilege for that storage group. The integrated catalog facility catalog used for the storage group must NOT contain an entry for the *n*th data set of the index.

stogroup-name must identify a storage group that exists at the current server and the privilege set must include SYSADM authority, SYSCTRL authority, or the USE privilege for the storage group.

If you omit PRIQTY, SECQTY, or ERASE from a USING STOGROUP clause for some partition, their values are given by the next USING STOGROUP clause that governs that partition: either a USING clause that is not in any PARTITION clause, or a default USING clause. DB2 assumes one and only one data set for each partition.

FREEPAGE *integer*
Specifies how often to leave a page of free space when index entries are created as the result of executing a DB2 utility or when creating an index for a table with existing rows. One free page is left for every *integer* pages. The value of *integer* can range from 0 to 255. The default is 0, leaving no free pages.

Do not specify FREEPAGE for an index on a declared temporary table.

PCTFREE *integer*

Determines the percentage of free space to leave in each nonleaf page and leaf page when entries are added to the index or index partition as the result of executing a DB2 utility or when creating an index for a table with existing rows. The first entry in a page is loaded without restriction. When additional entries are placed in a nonleaf or leaf page, the percentage of free space is at least as great as *integer*.

The value of *integer* can range from 0 to 99, however, if a value greater than 10 is specified, only 10 percent of free space will be left in nonleaf pages. The default is 10.

Do not specify PCTFREE for an index on a declared temporary table.

If the index is partitioned , the values of FREEPAGE and PCTFREE for a particular partition are given by the first of these choices that applies:

- The values of FREEPAGE and PCTFREE given in the PARTITION clause for that partition. Do not use more than one *free-specification* in any PARTITION clause.
- The values given in a *free-specification* that is not in any PARTITION clause.
- The default values FREEPAGE 0 and PCTFREE 10.

GBPCACHE

In a data sharing environment, specifies what index pages are written to the group buffer pool. In a non-data-sharing environment, the option is ignored unless the index is on a declared temporary table. Do not specify GBPCACHE for an index on a declared temporary table in either environment (data sharing or non-data-sharing).

CHANGED

Specifies that updated pages are written to the group buffer pool, when there is inter-DB2 R/W interest on the index or partition. When there is no inter-DB2 R/W interest, the group buffer pool is not used. Inter-DB2 R/W interest exists when more than one member in the data sharing group has the index or partition open, and at least one member has it open for update. GBPCACHE CHANGED is the default.

If the index is in a group buffer pool that is defined as GBPCACHE(NO), CHANGED is ignored and no pages are written to the group buffer pool.

ALL

Indicates that pages are written to the group buffer pool as they are read in from DASD.

Exception: In the case of a single updating DB2 subsystem when no other DB2 subsystems have any interest in the page set, no pages are written to the group buffer pool.

If the index is in a group buffer pool that is defined as GBPCACHE(NO), ALL is ignored and no pages are written to the group buffer pool.

NONE

Indicates that no pages are written to the group buffer pool. DB2 uses the group buffer pool only for cross-invalidation.

If the index is partitioned, the value of GBPCACHE for a particular partition is given by the first of these choices that applies:

1. The value of GBPCACHE given in the PARTITION clause for that partition. Do not use more than one *gbpcache-specification* in any PARTITION clause.

2. The value given in a *gbpcache-specification* that is not in any PARTITION clause.
3. GBPCACHE CHANGED is the default value.

DEFINE
Specifies when the underlying data sets for the index are physically created. The SPACE column in catalog table SYSINDEXPART is used to record the status of the data sets (undefined or allocated). If the DEFINE keyword is not specified, the define attribute is inherited from the current state of the base table space.

YES
The data sets are created when the index is created (the CREATE INDEX statement is executed).

NO The data sets are not created until data is inserted into the index.

DEFINE NO is applicable only for DB2-managed data sets (USING STOGROUP is specified). Use DEFINE NO especially when performance of the CREATE INDEX statement is important or DASD resource is constrained.

Do not use DEFINE NO on an index if you use a program outside of DB2 to propagate data into a table on which that index is defined. If you use DEFINE NO on an index of a table and data is then propagated into the table from a program that is outside of DB2, the index space data sets are allocated, but the DB2 catalog will not reflect this fact. As a result, DB2 treats the data sets for the index space as if they have not yet been allocated. The resulting inconsistency causes DB2 to deny application programs access to the data until the inconsistency is resolved.

DEFINE NO is ignored for user-managed data sets (USING VCAT is specified). DEFINE NO is also ignored if the index is being created on a table that is not empty or on an auxiliary table.

Do not specify DEFINE NO if the index is created on a base table that is involved in a clone relationship.

Do not specify DEFINE NO for an index on a declared temporary table.

COMPRESS NO or COMPRESS YES
Specifies whether compression for index data will be used. If the index is partitioned, the clause will apply to all partitions.

COMPRESS NO
Specifies that no index compression will be used.

COMPRESS NO is the default.

COMPRESS YES
Specifies that index compression will be used. The bufferpool that is used to create the index must be 8K, 16K, or 32K in size. The physical page size on disk will be 4K. The index compression will take place immediately.

Index compression is recommended for applications that do sequential insert operations with few or no delete operations. Random inserts and deletes can adversely effect compression. Index compress is also recommended for applications where the indexes are created primarily for scan operations.

INCLUDE NULL KEYS or EXCLUDE NULL KEYS
Specifies whether an index entry will be created when every key column contains the NULL value.

INCLUDE NULL KEYS
Specifies that an index entry will be created when every key column contains the NULL value.

INCLUDE NULL KEYS is the default.

EXCLUDE NULL KEYS
Specifies that no index entry will be created when every key column contains the NULL value. If any key column is not null the index entry will be created.

EXCLUDE NULL KEYS must not be specified with the following:

- **UNIQUE**
- **BUSINESS_TIME WITHOUT OVERLAPS**
- *XML-index-specification*
- *key-expression*
- **INCLUDE** (*column-name*)

EXCLUDE NULL KEYS must also not be specified if any of the columns that are identified by *column-name* are defined as NOT NULL, or if the index is defined as a partitioning index for use with index-controlled partitioning.

PARTITION BY RANGE
Specifies the partitioning index for the table, which determines the partitioning scheme for the data in the table.

PARTITION BY RANGE should only be specified if the table space is partitioned and the partitioning schema has not already been established.

PARTITION BY RANGE must not be specified if the index is an extended index, is defined with the BUSINESS_TIME WITHOUT OVERLAPS, or if the table is in a universal table space (ranged-partitioned or partition-by-growth table space).

partition-element
Specifies the range for each partition.

PARTITION *integer*
A PARTITION clause specifies the highest value of the index key in one partition of a partitioning index. In this context, highest means highest in the sorting sequences of the index columns. In a column defined as *ascending* (ASC), highest and lowest have their usual meanings. In a column defined as *descending* (DESC), the lowest actual value is highest in the sorting sequence.

If you use CLUSTER, and the table is contained in a partitioned table space, you must use exactly one PARTITION clause for each partition (defined with NUMPARTS on CREATE TABLESPACE). If there are *p* partitions, the value of *integer* must range from 1 through *p*.

The length of the highest value of a partition (also called the limit key) is the same as the length of the partitioning index.

ENDING AT(*constant***, MAXVALUE, or MINVALUE...)**
Specifies that this is the partitioning index and indicates how the data will be partitioned. The table space is marked complete after this partitioning index is created. You must use at least one value (*constant*, MAXVALUE, or MINVALUE) after ENDING AT in each PARTITION clause. You can use as many as there are columns in the key. The concatenation of all the values is the highest value of the key in the

corresponding partition of the index unless the VALUES statement was already specified when the table or previous index was created.

constant
Specifies a constant value with a data type that must conform to the rules for assigning that value to the column. If a string constant is longer or shorter than required by the length attribute of its column, the constant is either truncated or padded on the right to the required length. If the column is ascending, the padding character is X'FF'. If the column is descending, the padding character is X'00'. The precision and scale of a decimal constant must not be greater than the precision and scale of its corresponding column. A hexadecimal string constant (GX) cannot be specified.

MAXVALUE
Specifies a value greater than the maximum value for the limit key of a partition boundary (that is, all X'FF' regardless of whether the column is ascending or descending). If all of the columns in the partitioning key are ascending, a constant or the MINVALUE clause cannot be specified following MAXVALUE. After MAXVALUE is specified, all subsequent columns must be MAXVALUE.

MINVALUE
Specifies a value that is smaller than the minimum value for the limit key of a partition boundary (that is, all X'00' regardless of whether the column is ascending or descending). If all of the columns in the partitioning key are descending, a constant or the MAXVALUE clause cannot be specified following MAXVALUE. After MINVALUE is specified, all subsequent columns must be MINVALUE.

The key values are subject to the following rules:

- The first value corresponds to the first column of the key, the second value to the second column, and so on. Using fewer values than there are columns in the key has the same effect as using the highest or lowest values for the omitted columns, depending on whether they are ascending or descending.
- If a key includes a ROWID column or a column with a distinct type that is based on a ROWID data type, 17 bytes of the constant that is specified for the corresponding ROWID column are considered.
- The highest value of the key in any partition must be lower than the highest value of the key in the next partition.
- If the concatenation of all the values exceeds 255 bytes, only the first 255 bytes are considered.
- The highest value of the key in the last partition depends on how the table space is defined. For table spaces that are created without the LARGE or DSSIZE options, the values that you specify after VALUES are not enforced. The highest value of the key that can be placed in the table is the highest possible value of the key.

  For large partitioned table space, the values you specify are enforced. The value specified for the last partition is the highest value of the key that can be placed in the table. Any key values greater than the value that is specified for the last partition are out of range.

ENDING AT can be specified only if the ENDING AT clause was not specified on a previous CREATE or ALTER TABLE statement for the underlying table.

INCLUSIVE
Specifies that the specified range values are included in the data partition.

BUFFERPOOL *bpname*
Identifies the buffer pool that is to be used for the index. The *bpname* must identify an activated 4KB, 8KB, 16KB, or 32KB buffer pool and the privilege set must include SYSADM or SYSCTRL authority or the USE privilege for the buffer pool.

The default is the default 4KB buffer pool for indexes in the database. A buffer pool with a smaller size should be chosen for indexes with random insert patterns. A buffer pool with a larger size should be chosen for indexes with sequential insert patterns.

See "Naming conventions" on page 57 for more details about *bpname*. See *DB2 Command Reference* for a description of active and inactive buffer pools.

CLOSE
Specifies whether or not the data set is eligible to be closed when the index is not being used and the limit on the number of open data sets is reached.

YES
Eligible for closing. This is the default unless the index is on a declared temporary table.

NO Not eligible for closing.

If the limit on the number of open data sets is reached and there are no page sets that specify CLOSE YES to close, page sets that specify CLOSE NO will be closed.

For an index on a declared temporary table, DB2 uses CLOSE NO regardless of the value specified.

DEFER
Indicates whether the index is built during the execution of the CREATE INDEX statement. Regardless of the option specified, the description of the index and its index space is added to the catalog. If the table is determined to be empty and DEFER YES is specified, the index is neither built nor placed in a rebuild-pending status. Refer to *DB2 Administration Guide* for more information about using DEFER. Do not specify DEFER for an index on a declared temporary table or an auxiliary table.

NO The index is built. This is the default.

YES
The index is not built. If the table is populated, the index is placed in a rebuild-pending status and a warning message is issued; the index must be rebuilt by the REBUILD INDEX utility.

PIECESIZE *integer*
Specifies the maximum addressability of each data set for a non-partitioned index. The subsequent keyword K, M, or G, indicates the units of the value that is specified in *integer*.

K Indicates that the *integer* value is to be multiplied by 1024 to specify the maximum data set size in bytes. *integer* must be a power of two between 1 and 268435456.

M Indicates that the *integer* value is to be multiplied by 1048576 to specify the maximum data set size in bytes. *integer* must be a power of two between 1 and 262144.

G Indicates that the *integer* value is to be multiplied by 1073741824 to specify the maximum data set size in bytes. *integer* must be a power of two between 1 and 256.

Table 111 shows the valid values for the data set size, which depend on the size of the table space.

Table 111. Valid values of PIECESIZE clause

K units	M units	G units	Size attribute of table space
256K			
512K			
1024K	1M		
2048K	2M		
4096K	4M		
8192K	8M		
16384K	16M		
32768K	32M		
65536K	64M		
131072K	128M		
262144K	256M		
524288K	512M		
1048576K	1024M	1G	
2097152K	2048M	2G	
4194304K	4096M	4G	LARGE, DSSIZE 4G (or greater)
8388608K	8192M	8G	DSSIZE 8G (or greater)
16777216K	16384M	16G	DSSIZE 16G (or greater)
33554432K	32768M	32G	DSSIZE 32G (or greater)
67108864K	65536M	64G	DSSIZE 64G (or greater)
134217728K	131072M	128G	DSSIZE 128G (or greater)
268435456K	262144M	256G	DSSIZE 256G

PIECESIZE has no effect on primary and secondary space allocation as it is only a specification of the maximum amount of data that a data set can hold and not the actual allocation of storage.

If you change the PIECESIZE value with the ALTER INDEX statement, the index is put into REBUILD-pending status.

See the following for additional information:

- Number of pieces and maximum piece size for non-partitioned indexes and data-partitioned secondary indexes
- Choosing a value for PIECESIZE

COPY
Indicates whether the COPY utility is allowed for the index. Do not specify COPY for an index on a declared temporary table.

NO Does not allows full image or concurrent copies or the use of the RECOVER utility on the index. NO is the default.

YES
Allows full image or concurrent copies and the use of the RECOVER utility on the index.

Notes

Owner privileges:
The owner of the table has all table privileges (see "GRANT (table or view privileges)" on page 1721) with the ability to grant these privileges to others. For more information about ownership of the object, see "Authorization, privileges, permissions, masks, and object ownership" on page 70.

Effects of the DEFER clause:
If DEFER NO is implicitly or explicitly specified, the CREATE INDEX statement cannot be executed while a DB2 utility has control of the table space that contains the identified table.

If the identified table already contains data and if the index build is not deferred, CREATE INDEX creates the index entries for it. If the table does not yet contain data, CREATE INDEX creates a description of the index; the index entries are created when data is inserted into the table.

Errors evaluating the expressions for an index:
Errors that occur during the evaluation of an expression for an index are returned when the expression is evaluated. This can occur on an SQL data change statement, SELECT from an SQL data change statement, or the REBUILD INDEX utility. For example, the evaluation of the expression `10 / column_1` returns an error if the value in *column_1* is 0. The error is returned during CREATE INDEX processing if the table is not empty and contains a row with a value of zero in *column_1*, otherwise the error is returned during the processing of the insert or update operation when a row with a value of zero in *column_1* is inserted or updated.

Result length of expressions that return a string type:
If the result data type of *key-expression* is a string type and the result length cannot be calculated at bind time, the length is set to the maximum allowable length of that data type or the largest length that DB2 can estimate. In this case, the CREATE INDEX statement can fail because the total key length might exceed the limit of an index key.

For example, the result length of the expression `REPEAT('A', CEIL(1.1))` is VARCHAR(32767) and the result length of the expression `SUBSTR(DESCRIPTION,1,INTEGER(1.2))` is the length of the DESCRIPTION column. Therefore, a CREATE INDEX statement that uses any of these expressions as a *key-expression* might not be created because the total key length might exceed the limit of an index key.

Use of ASC or DESC on key columns:
There are no restrictions on the use of ASC or DESC for the columns of a parent key or foreign key. An index on a foreign key does not have to have the same ascending and descending attributes as the index of the corresponding parent key.

EBCDIC, ASCII, and UNICODE encoding schemes for an index:
An index has the same encoding scheme as its associated table.

Number of pieces and maximum piece size for non-partitioned indexes and data-partitioned secondary indexes
The largest amount of data that an index can hold is the maximum number of pieces for the index times the maximum amount of data that a piece can hold.

For a non-partitioned index, the maximum amount of data that an index can hold is defined by using the PIECESIZE parameter.

The default piece size for an index is as follows:

- 2 GB (PIECESIZE 2 G) for indexes of table spaces created without the LARGE or DSSIZE option
- 4 GB (PIECESIZE 4 G) for indexes of table spaces created with the LARGE or DSSIZE option
- 4 GB (PIECESIZE 4 G) for auxiliary indexes

The following tables list the maximum number of pieces and the default index piece size for various table spaces.

Table 112. Maximum number of pieces and the default index piece size for a partitioned table space that is created without the LARGE or DSSIZE clauses and has a NUMPART value of less than or equal to 64

Definition of partitioned table space (non-large), NUMPART value	Partitioned index		Non-partitioned index	
	Maximum number of pieces	Default index piece size	Maximum number of pieces	Default index piece size
NUMPARTS <= 16	16	4G	32	2G
NUMPART >= 17 but NUMPARTS <= 32	32	2G	32	2G
NUMPART >= 33	64	1G	32	2G

Table 113. Maximum number of pieces and the default index piece size for a partitioned table space that is created with the LARGE or DSSIZE clauses or has a NUMPART value of greater than or equal to 65

Definition of partitioned table space (large)	Partitioned index		Non-partitioned index	
	Maximum number of pieces	Default index piece size	Maximum number of pieces	Default index piece size
• LARGE clause - specified • DSSIZE clause - not specified	`MIN(4096, 2^32/(x/y))` - see note 1	4G	`MIN(4096, 2^32/(x/y))` - see note 2	4G
• LARGE clause - not specified • DSSIZE clause - not specified • NUMPARTS clause - greater than 64 but less than 256	`MIN(4096, 2^32/(x/y))` - see note 1	4G	`MIN(4096, 2^32/(x/y))` - see note 2	4G

Table 113. Maximum number of pieces and the default index piece size for a partitioned table space that is created with the LARGE or DSSIZE clauses or has a NUMPART value of greater than or equal to 65 (continued)

Definition of partitioned table space (large)	Partitioned index		Non-partitioned index	
	Maximum number of pieces	**Default index piece size**	**Maximum number of pieces**	**Default index piece size**
• LARGE clause - not specified • DSSIZE clause - specified **or** NUMPARTS clause - greater than or equal to 256	`MIN(4096, 2^32/(x/y))` - see note 1	`MIN(x, 2^32/(MIN(4096, 2^32/(x/y))) * z)` - see note 1 for values of *x* and *y*. *z* is the page size of the index.	`MIN(4096, 2^32/(x/y))` - see note 2	4G

Note:

1. For a partitioned index, the formula `MIN(4096, 2^32 / (x / y))`, determines the maximum number of partitions in the table space, where *x* and *y* have the following values:

   x is the DSSIZE of the table space

   y is the page size of the table space
2. For a non-partitioned index, the formula `MIN(4096, 2^32 / (x / y))`, determines the maximum number of pieces for the non-partitioned index, where *x* and *y* have the following values:

   x is the piece size of the index (stored in the PIECESIZE column of the SYSIBM.SYSINDEXES catalog table)

   y is the page size of the index (stored in the PGSIZE column of the SYSIBM.SYSINDEXES catalog table)

Table 114. Maximum number of pieces and the default index piece size for a non-partitioned table space

Type of non-partitioned table space	Maximum number of pieces	Default index piece size
non-segmented table space	32	2G
segmented table space	32	2G
LOB, auxiliary, or XML table space	32	4G

Choosing a value for PIECESIZE:

To choose a value for PIECESIZE, divide the size of the non-partitioned index by the number of data sets that you want. For example, to ensure that you have five data sets for the non-partitioned index, and your index is 10MB (and not likely to grow much), specify PIECESIZE 2 M. If your non-partitioned index is likely to grow, choose a larger value.

Remember that 32 data sets is the limit if the underlying table space is not defined as LARGE or with a DSSIZE parameter and that the limit is 4096 for objects with greater than 254 parts. For a non-partitioned index on a table space that is defined as LARGE or with a DSSIZE parameter, the maximum is MIN(4096, 2^{32} / (*index piece size*/*index page size*)).

Keep the PIECESIZE value in mind when you are choosing values for primary and secondary quantities. Ideally, the value of your primary quantity plus the secondary quantities should be evenly divisible into PIECESIZE.

Dropping an index:

Partitioning indexes can be dropped. If the table space is using index-controlled partitioning, the table space is converted to table-controlled partitioning. Secondary indexes that are not indexes on

auxiliary tables can be dropped simply by dropping the indexes. An empty index on an auxiliary table can be explicitly dropped; a populated index can be dropped only by dropping other objects. For details, see Dropping an index on an auxiliary table and an auxiliary table.

If the index is a unique index that enforces a primary key, unique key, or referential constraint, the constraint must be dropped before the index is dropped. See "DROP" on page 1609.

Unique indexes and enforcement of UNIQUE or PRIMARY KEY specifications for a table:

A table requires a unique index (that is not defined as UNIQUE WHERE NOT NULL) if you use the UNIQUE or PRIMARY KEY clause in the CREATE or ALTER TABLE statements, or if there is a ROWID column that is defined as GENERATED BY DEFAULT. DB2 implicitly creates those unique indexes if the table space is explicitly created and the CREATE or ALTER TABLE statement is processed by the schema processor or if the table space is implicitly created; otherwise, you must explicitly create them. If any of the unique indexes that must be explicitly defined do not exist, the definition of the table is incomplete, and the following rules apply:

- Let K denote a key for which a required unique index does not exist and let *n* denote the number of unique indexes that remain to be created before the definition of the table is complete. (For a new table that has no indexes, K is its primary key or any of the keys defined in the CREATE or ALTER TABLE statement as UNIQUE and *n* is the number of such keys. After the definition of a table is complete, an index cannot be dropped if it is enforcing a primary key or unique key.)
- The creation of the unique index reduces *n* by one if the index key is identical to K. The keys are identical only if they have the same columns in the same order.
- If *n* is now zero, the creation of the index completes the definition of the table.
- If K is a primary key, the description of the index indicates that it is a primary index. If K is not a primary key, the description of the index indicates that it enforces the uniqueness of a key defined as UNIQUE in the CREATE or ALTER TABLE statement.

A unique index cannot be created on a materialized query table.

Unique indexes and XML columns:

If the index is an XML index on a unique XML column, the uniqueness applies to values of the specified pattern across all documents of that column, and the uniqueness is enforced on the value after the value is cast to the specified SQL data type. Because the data type conversion might result in a loss of precision and normalization, multiple values that appear unique in the XML document might still result in duplicate errors. If the index is defined using an expression, the uniqueness is enforced against the values that are stored in the index, not against the original values of the columns. The **WHERE NOT NULL** specification is ignored with a warning if XMLPATTERN is also specified, and the index is treated as if **UNIQUE** had been specified.

Defining an XML index using an XPath pattern-expression that includes functions:

An XPath *pattern-expression* that includes functions (including fn:exists() or fn:upper-case()) will have two parts. The first part is referred to as the *context step* and specifies the XPath of the element node or attribute node

for which an index entry will be created (the element or attributes NodeID will be included in the index). The context step follows the same syntax as the XPath *pattern-expression* for an XML index, except that for fn:exists() it has to specify an element node, and for fn:upper-case() it has to specify an element node or an attribute node.

The second part is referred to as the *function expression step* and specifies the fn:exists() or fn:upper-case() XPath function. The function expression step is the right-most part of an XPath *pattern-expression*. For each node specified by the context step, the function expression step specifies the key value for the index. For example, in the XPath *pattern-expression* `/purchaseOrder/items/item/fn:exists(shipDate)`, the context step is `/purchaseOrder/items/item`, and the function expression step is `fn:exists(shipDate)`.

Use of PARTITIONED keyword:
When a partitioned index is created and no additional keywords are specified, the index is non-partitioned. If the keyword PARTITIONED is specified, the index is partitioned. If PARTITION BY RANGE is specified, the index is both data-partitioned and key-partitioned because it is defined on the partitioning columns of the table. Any index on a partitioned table space that does not meet the definition of a partitioning index is a secondary index. When a secondary index is created and no additional keywords are specified, the secondary index is non-partitioned (NPSI). If the keyword PARTITIONED is specified, the index is a data-partitioned secondary index (DPSI).

Creating a partitioning index for a table created without partition boundaries:
When a table is created without specifying partition boundaries using the ENDING AT clause, the table is incomplete until a partitioning index is created. The first index that is created for a table must specify both the PARTITION and the ENDING AT clauses.

When the PARTITION clause is specified while creating an index, either the PARTITIONED clause, or the ENDING AT clause must also be specified.

Considerations for tables that are involved in a clone relationship:
If an index is created on a base table that is involved in a clone relationship, an index with the same name is also created on the clone table. The index on the clone table will be placed in rebuild-pending status unless the clone table is empty when the index is created.

Considerations for tables that contain a row change timestamp column:
To create an index that refers to a row change timestamp column in the table, values must already exist in the column for all rows. Values are stored in row change timestamp columns whenever a row is inserted or updated in the table. If the row change timestamp column is added to an existing table that contains rows, the values for the row change timestamp column is not materialized and stored at the time of the ALTER TABLE statement. Values are materialized for these rows when they are updated, or when a REORG or a LOAD REPLACE utility is run on the table or table space.

Restriction on table spaces when there are pending changes to the definition:
A CREATE INDEX statement is not allowed if there are pending changes to the definition of the table space or to any objects in the table space. In

addition, an index that references an expression cannot be created on a table where the inline length of a LOB column has been changed and the table space has not been reorganized.

Effects of DEFINE NO and INCLUDE NULL KEYS or EXCLUDE NULL KEYS:
When INCLUDE NULL KEYS is specified (implicitly or explicitly) with DEFINE NO and the table that is being indexed is populated, a warning is returned, the index is created, and the data set is defined. When EXCLUDE NULL KEYS is specified, it is possible that the data set will not be defined if the all rows for the table that is being indexed contain the NULL value for all key columns. The index will be empty after the CREATE INDEX statement. However, if DEFINE NO is specified with EXCLUDE NULL KEYS a warning is returned.

Creating indexes on DB2 catalog tables:
To avoid problems during recovery of the DB2 catalog, any user-created indexes on DB2 catalog tables should be created in user-managed data sets.

For details on creating indexes on catalog tables, see "SQL statements allowed on the catalog" on page 2113.

EA-enabled index data sets:
If an index is created for an EA-enabled table space, the data sets for the index must be set up to belong to a DFSMS data class that has the extended format and extended addressability attributes.

Alternative syntax and synonyms:
To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords when creating a partitioned index:

- **PART** *integer* **VALUES** as an alternative syntax for **PARTITION** *integer* **ENDING**. The **PARTITION BY RANGE** keyword that precedes the *partition-element* clause is optional.

Although these keywords are supported as alternatives, they are not the preferred syntax.

Examples

Example 1: Create a unique index, named DSN8B10.XDEPT1, on table DSN8B10.DEPT. Index entries are to be in ascending order by the single column DEPTNO. DB2 is to define the data sets for the index, using storage group DSN8G110. Each data set should hold 1 megabyte of data at most. Use 512 kilobytes as the primary space allocation for each data set and 64 kilobytes as the secondary space allocation. These specifications enable each data set to be extended up to 8 times before a new data set is used—512KB + (8*64KB)= 1024KB. Make the index padded.

The data sets can be closed when no one is using the index and do not need to be erased if the index is dropped.

```
CREATE UNIQUE INDEX DSN8B10.XDEPT1
  ON DSN8B10.DEPT
    (DEPTNO ASC)
  PADDED
  USING STOGROUP DSN8G110
    PRIQTY 512
    SECQTY 64
```

```
    ERASE NO
  BUFFERPOOL BP1
  CLOSE YES
  PIECESIZE 1M;
```

For the above example, the underlying data sets for the index will be created immediately, which is the default (DEFINE YES). Assuming that table DSN8B10.DEPT is empty, if you wanted to defer the creation of the data sets until data is first inserted into the index, you would specify DEFINE NO instead of accepting the default behavior. Specifying PADDED ensures that the varying-length character string columns in the index are padded with blanks.

Example 2: Create a cluster index, named XEMP2, on table EMP in database DSN8B10. Put the entries in ascending order by column EMPNO. Let DB2 define the data sets for each partition using storage group DSN8G110. Make the primary space allocation be 36 kilobytes, and allow DB2 to use the default value for SECQTY, which for this example is 12 kilobytes (3 times 4KB). If the index is dropped, the data sets need not be erased.

There are to be 4 partitions, with index entries divided among them as follows:
Partition 1: entries up to H99
Partition 2: entries above H99 up to P99
Partition 3: entries above P99 up to Z99
Partition 4: entries above Z99

Associate the index with buffer pool BP1 and allow the data sets to be closed when no one is using the index. Enable the use of the COPY utility for full image or concurrent copies and the RECOVER utility.

```
CREATE INDEX DSN8B10.XEMP2
  ON DSN8B10.EMP
    (EMPNO ASC)
  USING STOGROUP DSN8G110
    PRIQTY 36
    ERASE NO
    CLUSTER
    PARTITION BY RANGE
    (PARTITION 1 ENDING AT('H99'),
     PARTITION 2 ENDING AT('P99'),
     PARTITION 3 ENDING AT('Z99'),
     PARTITION 4 ENDING AT('999'))
  BUFFERPOOL BP1
  CLOSE YES
  COPY YES;
```

Example 3: Create a secondary index, named DSN8B10.XDEPT1, on table DSN8B10.DEPT. Put the entries in ascending order by column DEPTNO. Assume that the data sets are managed by the user with catalog name DSNCAT and each data set is to hold 1GB of data, at most, before the next data set is used.

```
CREATE UNIQUE INDEX DSN8B10.XDEPT1
  ON DSN8B10.DEPT
    (DEPTNO ASC)
  USING VCAT DSNCAT
  PIECESIZE 1048576K;
```

Example 4: Assume that a column named EMP_PHOTO with a data type of BLOB(110K) was added to the sample employee table for each employee's photo and auxiliary table EMP_PHOTO_ATAB was created in LOB table space

DSN8D11A.PHOTOLTS to store the BLOB data for the column. Create an index named XPHOTO on the auxiliary table. The data sets are to be user-managed with catalog name DSNCAT.

```
CREATE UNIQUE INDEX DSN8B10.XPHOTO
   ON DSN8B10.EMP_PHOTO_ATAB
   USING VCAT DSNCAT
   COPY YES;
```

In this example, no columns are specified for the key because auxiliary indexes have implicitly generated keys.

CREATE MASK

The CREATE MASK statement creates a column mask at the current server. A *column mask* is used for column access control and specifies the value that should be returned for a specified column.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

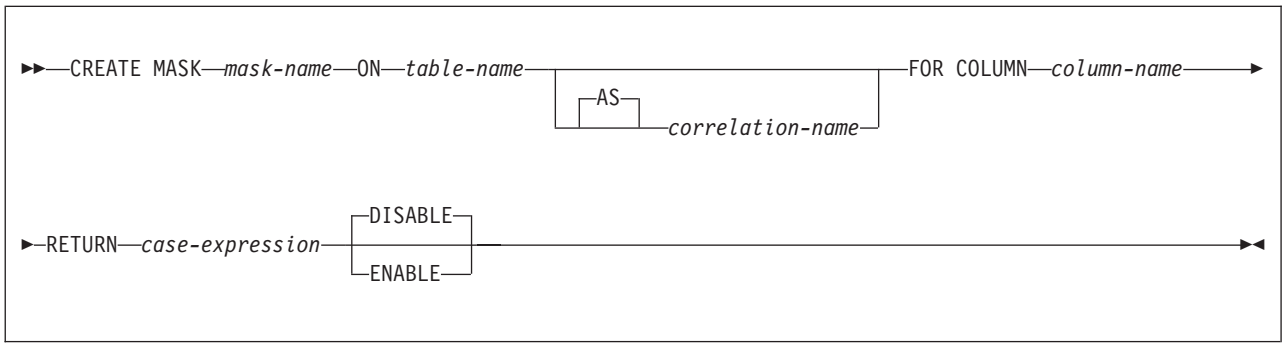
The privilege set that is defined below must include the following authority:

SECADM authority

SECADM authority can create a column mask in any schema. Additional privileges are not needed to reference other objects in the mask definition. For example, the SELECT privilege is not needed query a table, and the EXECUTE privilege is not needed to invoke a user-defined function.

Privilege set: If the statement is embedded in an application program, the privilege set is the set of privileges that are held by the owner of the package. If the statement is dynamically prepared, the privilege set is the set of privileges that are held by the SQL authorization ID of the process. However, if the process is running in a trusted context that is defined with the ROLE AS OBJECT OWNER AND QUALIFIER clause, the privilege set is the set of privileges that are held by the role that is in effect.

Syntax



Description

mask-name

Specifies the names the column mask. The name, including the implicit or explicit qualifier, must not identify a column mask or a row permission that already exists at the current server.

ON *table-name*

Identifies the table for which the column mask is created. The name must identify a table that exists at the current server. It must not identify any of the following objects:

- An auxiliary table

- A created or declared temporary table
- A view
- A catalog table
- An alias
- A synonym
- A materialized query table or table that is directly or indirectly referenced in the definition of a materialized query table
- A table that was implicitly created for an XML column
- A table that contains a period
- A history table
- An archive-enabled table
- An archive table

correlation-name
Specifies a correlation name that can be used within *case-expression* to designate the table. For information about *correlation-name*, see "Correlation names" on page 209.

FOR COLUMN *column-name*
Identifies the column to which the mask applies. *column-name* must be an unqualified name that identifies a column of the specified table. A mask must not already exist for the column. The column must not be:

- a LOB column or a distinct type column that is based on a LOB
- an XML column
- defined with a FIELDPROC
- a Unicode column in an EBCDIC table

RETURN *case-expression*
Specifies a CASE expression that determines the value that is returned for the column. The result of the CASE expression is returned in place of the column value in a row. The result data type, null attribute, data length, subtype, encoding scheme, and CCSID of the CASE expression must be identical to those attributes of the column that is specified by *column-name*. If the data type of *column-name* is a user-defined data type, the result data type of the CASE expression must be the same user-defined type. The CASE expression must not reference any of the following objects:

- A remote object
- The table for which the column mask is being defined
- A created global temporary table or a declared global temporary table
- An auxiliary table
- A table that was implicitly created for an XML column
- A column that is defined with a FIELDPROC
- A LOB column or a distinct type column that is based on a LOB
- An XML column
- A select list notation * or *name*.* in the SELECT clause
- A table function
- A collection-derived table (UNNEST)
- A user-defined function that is defined as not secure
- A function that is not deterministic or that has an external action or is defined with the MODIFIES SQL DATA option

- An aggregate function, unless it is specified in a subquery
- A built-in table function
- An XMLTABLE table function
- An XMLEXISTS predicate
- An OLAP specification
- A ROW CHANGE expression
- A sequence reference
- A host variable, SQL variable, SQL parameter, or trigger transition variable
- A parameter marker
- A table reference that contains a period specification
- A view that includes any of the preceding restrictions in its definition

The encoding scheme of the specified table is used to evaluate the CASE expression. Tables and language elements that require multiple encoding scheme evaluation must not be referenced in the CASE expression. See "Determining the encoding scheme and CCSID of a string" on page 47 for language elements that require multiple evaluation.

If the CASE expression references tables for which row or column access control is active, access controls for those tables are not cascaded.

DISABLE or ENABLE
Specifies that the column mask is to be enabled or disabled for column access control.

DISABLE
Specifies that the column mask is to be disabled for column access control. The column mask will remain disabled regardless of whether column access control is activated for the table.

DISABLE is the default.

ENABLE
Specifies that the column mask is to be enabled for column access control. If column access control is not currently active for the table, the column mask will become enabled when column access control is activated for the table. If column access control is currently active for the table, the column mask becomes enabled immediately and all packages and statements in the dynamic statement cache that reference the table are invalidated.

Notes

How column masks affect queries:
The application of enabled column masks does not interfere with the operations of other clauses within the statement such as the WHERE, GROUP BY, HAVING, SELECT DISTINCT, or ORDER BY. The rows that are returned in the final result table remain the same, except that the values in the resulting rows might have been masked by the column masks. As such, if the masked column also appears in an ORDER BY clause with a *sort-key* expression, the order is based on the original values of the column and the masked values in the final result table might not reflect that order. Similarly, the masked values might not reflect the uniqueness enforced by a SELECT DISTINCT statement. If the masked column is embedded in an expression, the result of the expression might become different because the column mask is applied on the column before the expression evaluation can take place. For example, a column mask on column SSN might change the result of the aggregate function

`COUNT(DISTINCT SSN)` because the DISTINCT operation is performed on the masked values. However, if the expression in the query is the same as the expression that is used to mask the column value in the definition of the column mask, the result of the expression might remain unchanged. For example, the expression in the query is `'XXX-XX-' || SUBSTR(SSN, 8, 4)` and the same expression appears in the definition of the column mask. In this particular example, you can replace the expression in the query with column SSN to avoid the same expression being evaluated twice.

Conflicts between the definition of a column mask and SQL:
A column mask is created as a stand alone object, without knowing all of the contexts in which it might be used. To mask the value of a column in the final result table, the definition of the column mask is merged into a query by DB2. When the definition of the column mask is brought into the context of the statement, it might conflict with certain SQL semantics in the statement. Therefore, in some situations, the combination of the statement and the application of the column mask can return an error. When this happens, either the statement needs to be modified or the column mask must be dropped or re-created with a different definition. See "ALTER TABLE" on page 984 for those situations in which a bind time error might be issued for the statement.

Column masks and null columns:
If the column is not nullable, the definition of its column mask will not, most likely, consider a null value for the column. After the column access control is activated for the table, if the table is the null-padded table in an outer join, the value of he column in the final result table might be a null. To ensure that the column mask can mask a null value, if the table is the null-padded table in an outer join, DB2 will add "**WHEN** *target-column* **IS NULL THEN NULL**" as the first WHEN clause to the column mask definition. This forces a null value to always be masked as a null value. For a nullable column, this removes the ability to mask a null value as something else. Example 5 shows this added WHEN clause.

Column mask values for SQL data change statements
When columns are used to derive new values for an INSERT, UPDATE, MERGE, or a SET *transition-variable* assignment statement, the original values of the column, not the masked values, are used to derive the new values. If the columns have column masks, those column masks are applied to ensure that the evaluation of the access control rules at run time masks the column to itself, not to a constant or an expression. This is to ensure that the masked values are the same as the original column values. If a column mask does not mask the column to itself, the existing row is not updated or the new row is not inserted and an error is returned at run time. The rules that are used to apply column masks in order to derive the new values follow the same rules for the final result table of a query.

Column masks that are created before column access control is activated:
The CREATE MASK statement is an independent statement that can be used to create a column access control mask before column access control is activated for a table. The only requirement is that the table and the columns exist before the mask is created. Multiple column masks can be created for a table but a column can have one mask only.

The definition of a mask is stored in the DB2 catalog. Dependency on the table for which the mask is being created and dependencies on other objects referenced in the definition are recorded. No package or dynamic cached statement is invalidated. A column mask can be created as enabled

or disabled for column access control. An enabled column mask does not take effect until the ALTER TABLE statement with the ACTIVATE COLUMN ACCESS CONTROL clause is used to activate column access control for the table. SECADM authority is required to issue such an ALTER TABLE statement. A disabled column mask remains ineffective even when column access control is activated for the table. The ALTER MASK statement can be used to alter between ENABLE and DISABLE.

After column access control is activated for a table, when the table is referenced in a data manipulation statement, all enabled column masks that have been created for the table are implicitly applied by DB2 to mask the values returned for the columns referenced in the final result table of the queries or to determine the new values used in the data change statements.

Creating column masks before activating column access control for a table is the recommended sequence to avoid multiple invalidations of packages and dynamic cached statements that reference the table.

Column masks that are created after column access control is activated:
The enabled column masks become effective as soon as they are committed. All the packages and dynamic cached statements that reference the table are invalidated. Thereafter, when the table is referenced in a data manipulation statement, all enabled column masks are implicitly applied by DB2 to the statement. Any disabled column mask remains ineffective even when column access control is activated for the table.

No cascaded effect when column or row access control enforced tables are referenced in column mask definitions:
A column mask definition may reference tables and columns that are currently enforced by row or column access control. Access control from those tables and columns are ignored when the table for which the column mask is being created is referenced in a data manipulation statement.

Multiple column masks and row permissions sharing the same environment variables:
Multiple column masks and row permissions can be created for a table. They must use the same set of environment variables. The set of environment variables is determined when the first column mask or row permission is created for the table.

The catalog table SYSENVIRONMENT contains the list of environment variables. The following table shows which environment variable must be the same among the multiple column masks and row permissions.

Table 115. Environment Variables in SYSIBM.SYSENVIRONMENT

Environment variables shown as SYSENVIRONMENT columns	Description	Static create statement	Dynamic create statement	Must be the same among multiple column masks and row permissions?
ENVID	Internal identifier of the environment	Assigned by DB2	Assigned by DB2	Yes
CURRENT_SCHEMA	The qualifier used to qualify unqualified objects such as tables, views. etc.	Package owner	Value of CURRENT_SCHEMA special register	Yes

Table 115. Environment Variables in SYSIBM.SYSENVIRONMENT (continued)

Environment variables shown as SYSENVIRONMENT columns	Description	Static create statement	Dynamic create statement	Must be the same among multiple column masks and row permissions?
PATHSCHEMAS	The schema path used to qualify unqualified object such as user-defined functions and CAST functions for user-defined data types.	PATH bind option	Value of CURRENT_PATH special register	Yes
APPLICATION_ ENCODING_ CCSID	The CCSID of the application environment	ENCODING bind option	CURRENT APPLICATION ENCODING SCHEME special register	Yes
ORIGINAL_ ENCODING_ CCSID	The original CCSID of the statement text string	CCSID(n) pre-compiler option or EBCDIC CCSID on DSNTIPF installation panel	CCSID based on DEF ENCODING SCHEME on DSNTIPF installation panel	Yes
DECIMAL_POINT	The decimal point indicator	COMMA or PERIOD precompiler option or DECIMAL POINT IS on DSNTIPF installation panelv	DECIMAL POINT IS on DSNTIPF installation panel	Yes
MIN_DIVIDE_SCALE	The minimum divide scale	MINIMUM DIVIDE SCALE on DSNTIP4 installation panel	MINIMUM DIVIDE SCALE on DSNTIP4 installation panelv	Yesv
STRING_DELIMITER	The string delimiter that is used in COBOL string constants	APOST precompiler option or STRING DELIMITER on DSNTIPF installation panel	STRING DELIMITER on DSNTIPF installation panel	No
SQL_ STRING_ DELIMITER	The SQL string delimiter that is used in constants	APOSTSQL pre-compiler option or SQL STRING DELIMITER on DSNTIPF installation panel	SQL STRING DELIMITER on DSNTIPF installation panel	Yes
MIXED_DATA	Uses mixed DBCS data	MIXED DATA on DSNTIPF installation panel	MIXED DATA on DSNTIPF installation panel	Yes
DECIMAL_ ARITHMETIC	The rules that are to be used for CURRENT PRECISION and when both operands in a decimal operation have a precision of 15 or less.	DEC(15) or DEC(31) precompiler option or DECIMAL ARITHMETIC on DSNTIP4 installation panel	DECIMAL ARITHMETIC on DSNTIP4 installation panel	Yes

Table 115. Environment Variables in SYSIBM.SYSENVIRONMENT (continued)

Environment variables shown as SYSENVIRONMENT columns	Description	Static create statement	Dynamic create statement	Must be the same among multiple column masks and row permissions?
DATE_FORMAT	The date format	DATE pre-compiler option or DATE FORMAT on DSNTIP4 installation panel	DATE FORMAT on DSNTIP4 installation panel	Yes
TIME_FORMAT	The time format	TIME pre-compiler option or TIME FORMAT on DSNTIP4 installation panel	TIME FORMAT on DSNTIP4 installation panel	Yes
FLOAT_FORMAT	The floating point format	FLOAT (S390 | IEEE) pre-compiler option or default of FLOAT S390	Default of FLOAT S390	No
HOST_LANGUAGE	The host language	HOST pre-compiler option or LANGUAGE DEFAULT on DSNTIPF installation panel	LANGUAGE DEFAULT on DSNTIPF installation panel	No
CHARSET	The character set	CCSID(n) pre-compiler option or EBCDIC CCSID on DSNTIPF installation panel	EBCDIC CCSID on DSNTIPF installation panel	No
FOLD	FOLD is only applicable when HOST_LANGUAGE is C or CPP. Otherwise FOLD is blank.	HOST(C(FOL D) precompiler option or default of NO FOLD	default of NO FOLD	No
ROUNDING	The rounding mode that is used when arithmetic and casting operations are performed on DECFLOAT data.	ROUNDING bind option	CURRENT DECFLOAT ROUNDING MODE special register	Yes

Note: In a data sharing environment, if a separate DSNHDECP module is provided for each member of the group, the DSNHDECP settings for each environment variable should be the same in all members of the data sharing group, otherwise an error might be issued when multiple column masks or row permissions are created.

Ordinary SQL identifiers specified in a static CREATE MASK statement in a COBOL application:

If the CREATE MASK statement is a static statement in a COBOL application, the ordinary SQL identifiers used in the column mask definition must not follow the rules for naming COBOL words. They must follow the rules for naming "SQL identifiers" on page 55. For example, the COBOL word 1ST-TIME is not allowed as an ordinary SQL identifier in a column mask definition; change it to FIRST_TIME or put it in the delimiters.

Encoding scheme and CCSIDs of the data manipulation statement after column masks are applied:

The encoding scheme and CCSIDs of the data manipulation statement is not affected by the column masks that are implicitly applied by DB2 for the column access control. The column mask definition is evaluated using its target table's encoding scheme and CCSIDs.

Consideration for DB2 limits:

If the data manipulation statement already approaches some DB2 limits in the statement, it should be noted that the more enabled column masks and enabled row permissions are created, the more likely they would impact some limits. For example, they may cause the statement to exceed the maximum total length (32600 bytes) of columns of a query operation requiring sort and evaluating aggregate functions (MULTIPLE DISTINCT and GROUP BY). This is because the enabled column mask and enabled row permission definitions are implicitly merged into the statement when the table is referenced in a data manipulation statement. See "Limits in DB2 for z/OS" in SQL Reference for the limits of a statement.

Restrictions involving pending definition changes:

CREATE MASK is not allowed if the mask is defined on a table or references a table that has pending definition changes.

Examples

Example 1:

After column access control is activated for table EMPLOYEE, Paul from the payroll department can see the social security number of the employee whose employee number is 123456. Mary who is a manager can see the last four characters only of the social security number. Peter who is neither cannot see the social security number.

```
CREATE MASK SSN_MASK ON EMPLOYEE
   FOR COLUMN SSN RETURN
     CASE
          WHEN (VERIFY_GROUP_FOR_USER(SSESSION_USER,'PAYROLL') = 1)
         THEN SSN
          WHEN (VERIFY_GROUP_FOR_USER(SESSION_USER,'MGR') = 1)
           THEN 'XXX-XX-' || SUBSTR(SSN,8,4)
          ELSE NULL
     END
   ENABLE;

COMMIT;

ALTER TABLE EMPLOYEE
    ACTIVATE COLUMN ACCESS CONTROL;

COMMIT;

SELECT SSN FROM EMPLOYEE
    WHERE EMPNO = 123456;
```

Example 2:

In the SELECT statement, column SSN is embedded in an expression that is the same as the expression used in the column mask SSN_MASK. After column access control is activated for table EMPLOYEE, the column mask SSN_MASK is applied to column SSN in the SELECT statement. For this particular expression, the SELECT statement produces the same result as before column access control is activated for all users. The user can replace the expression in the SELECT statement with column SSN to avoid the same expression gets evaluated twice.

```
CREATE MASK SSN_MASK ON EMPLOYEE
    FOR COLUMN SSN RETURN
      CASE
           WHEN (1 = 1)
            THEN 'XXX-XX-' || SUBSTR(SSN,8,4)
           ELSE NULL
      END
    ENABLE;

COMMIT;

ALTER TABLE EMPLOYEE
    ACTIVATE COLUMN ACCESS CONTROL;

COMMIT;

SELECT 'XXX-XX-' || SUBSTR(SSN,8,4) FROM EMPLOYEE
    WHERE EMPNO = 123456;
```

Example 3:

A state government conducted a survey for the library usage of the households in each city. Fifty households in each city were sampled in the survey. Each household was given an option, opt-in or opt-out, whether to show their usage in any reports generated from the result of the survey.

A SELECT statement is used to generate a report to show the average hours used by households in each city. Column mask CITY_MASK is created to mask the city name based on the opt-in or opt-out information chosen by the sampled households. However, after column access control is activated for table LIBRARY_ USAGE, the SELECT statement receives a bind time error. This is because column mask CITY_MASK references another column LIBRARY_OPT and LIBRARY_OPT does not identify a grouping column.

```
CREATE MASK CITY_MASK ON LIBRARY_USAGE
    FOR COLUMN CITY RETURN
      CASE
           WHEN (LIBRARY_OPT = 'OPT-IN')
            THEN CITY
           ELSE ' '
      END
    ENABLE;

COMMIT;

ALTER TABLE LIBRARY_USAGE
   ACTIVATE COLUMN ACCESS CONTROL;

COMMIT;

SELECT CITY, AVG(LIBRARY_TIME) FROM LIBRARY_USAGE
   GROUP BY CITY;
```

Example 4:

Employee with EMPNO 123456 earns bonus $8000 and salary $80000 in May. When the manager retrieves his salary, the manager receives his salary, not the null value. This is because of no cascaded effect when column mask SALARY_MASK references column BONUS for which column mask BONUS_MASK is defined.

```
CREATE MASK SALARY_MASK ON EMPLOYEE
   FOR COLUMN SALARY RETURN
       CASE
            WHEN (BONUS < 10000)
             THEN SALARY
            ELSE NULL
```

```
       END
   ENABLE;

COMMIT;

CREATE MASK BONUS_MASK ON EMPLOYEE
   FOR COLUMN BONUS RETURN
       CASE
            WHEN (BONUS > 5000)
             THEN NULL
            ELSE BONUS
       END
   ENABLE;

COMMIT;

ALTER TABLE EMPLOYEE
    ACTIVATE COLUMN ACCESS CONTROL;

COMMIT;

SELECT SALARY FROM EMPLOYEE
    WHERE EMPNO = 123456;
```

Example 5:

This example shows DB2 adds "WHEN target-column IS NULL THEN NULL" as the first WHEN clause to the column mask definition then merges the column mask definition into the statement.

```
CREATE EMPLOYEE (EMPID INT,
                 DEPTID CHAR(8),
                 SALARY DEC(9,2) NOT NULL,
                 BONUS DEC(9,2));

CREATE MASK SALARY_MASK ON EMPLOYEE
    FOR COLUMN SALARY RETURN
       CASE
            WHEN SALARY < 10000
             THEN CAST(SALARY*2 AS DEC(9,2))
            ELSE COALESCE(CAST(SALARY/2 AS DEC(9,2)), BONUS)
       END
    ENABLE;

COMMIT;

CREATE MASK BONUS_MASK ON EMPLOYEE
    FOR COLUMN BONUS RETURN
      CASE
          WHEN BONUS > 1000
           THEN BONUS
          ELSE NULL
      END
    ENABLE;

COMMIT;

ALTER TABLE EMPLOYEE
    ACTIVATE COLUMN ACCESS CONTROL;

COMMIT;

SELECT SALARY FROM DEPT
    LEFT JOIN EMPLOYEE ON DEPTNO = DEPTID;

/* When SALARY_MASK is merged into the above statement,
 * 'WHEN SALARY IS NULL THEN NULL' is added as the
 * first WHEN clause, as follows:
```

```
 */

SELECT CASE WHEN SALARY IS NULL THEN NULL
            WHEN SALARY < 10000 THEN CAST(SALARY*2 AS DEC(9,2))
            ELSE COALESCE(CAST(SALARY/2 AS DEC(9,2)), BONUS)
       END SALARY
       FROM DEPT
         LEFT JOIN EMPLOYEE ON DEPTNO = DEPTID;
```

CREATE PERMISSION

The CREATE PERMISSION statement creates a row permission for row access control at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

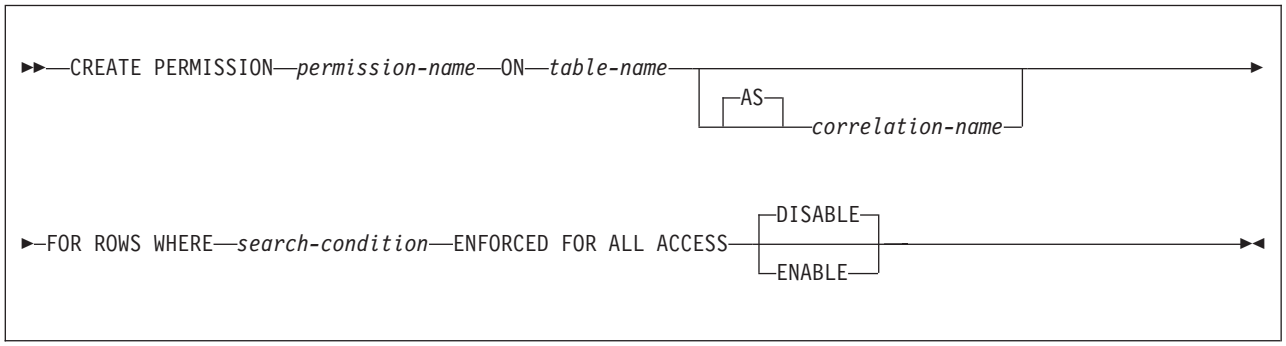
The privilege set that is defined below must include the following authority:

- SECADM authority

SECADM authority can create a row permission in any schema. Additional privileges are not needed to reference other objects in the permission definition. For example, the SELECT privilege is not needed to retrieve from a table, and the EXECUTE privilege is not needed to invoke a user-defined function.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the package. If the statement is dynamically prepared, the privilege set is the set of privileges that are held by the SQL authorization ID of the process. However, if it is running in a trusted context defined with the ROLE AS OBJECT OWNER AND QUALIFIER clause, the privilege set is the set of privileges that are held by the role in effect.

Syntax



Description

permission-name
Names the row permission for row access control. The name, including the implicit or explicit qualifier, must not identify a row permission or a column mask that already exists at the current server

ON *table-name*
Identifies the table on which the row permission is created. The name must identify a table that exists at the current server. It must not identify any of the following objects:

- An auxiliary table
- A created or declared temporary table
- A view

- A catalog table
- An alias
- A synonym
- A materialized query table or table that is directly or indirectly referenced in the definition of a materialized query table
- A table that was implicitly created for an XML column
- A table that contains a period
- A history table
- An archive-enabled table
- An archive table
- A table that has a security label column.

correlation-name
Can be used within *search-condition* to designate the table. For the explanation of *correlation-name*, see "Correlation names" on page 209.

FOR ROWS WHERE
Indicates that a row permission is created. A row permission specifies a search condition under which rows of the table can be accessed.

search-condition
Specifies a condition that can be true, false, or unknown for a row of the table. *search-condition* follows the same rules that are used by the search condition in a WHERE clause of a subselect. In addition, the search condition must not reference any of the following objects:

- A remote object
- The table for which the row permission is being defined
- A table that has a security label column
- A created global temporary table or a declared global temporary table
- An auxiliary table
- A table that was implicitly created for an XML column
- A collection-derived table (UNNEST)
- A table function
- A host variable, SQL variable, SQL parameter, or trigger transition variable
- A user-defined function that is defined as not secure
- A function that is not deterministic or that has an external action or is defined with the MODIFIES SQL DATA option
- A parameter marker
- A column that is defined with a FIELDPROC
- A LOB column or a distinct type column that is based on a LOB
- An XML column
- A Unicode column in an EBCDIC table
- An XMLEXISTS predicate
- An OLAP specification
- A ROW CHANGE expression
- A sequence reference
- A select list notation `*` or *name*`.*` in the SELECT clause
- A table reference that contains a period specification
- A view that includes any of the preceding restrictions in its definition

The encoding scheme of the table is used to evaluate the *search-condition*. Tables and language elements that require multiple encoding scheme evaluation must not be referenced in the *search-condition*. See "Determining the encoding scheme and CCSID of a string" on page 47 for those language elements.

If the *search-condition* references tables for which row or column access control is activated, access control from those tables is not cascaded.

ENFORCED FOR ALL ACCESS
Specifies that the row permission applies to all references of the table. If row access control is activated for the table, when the table is referenced in a data manipulation statement, DB2 implicitly applies the row permission to control the access of the table. If the reference of the table is for a fetch operation such as SELECT, the application of the row permission determines what set of rows can be retrieved by the user who requested the fetch operation. If the reference of the table is for a data change operation such as INSERT, the application of the row permission determines whether all rows to be changed are insertable or updatable by the user who requested the data change operation.

DISABLE or ENABLE
Specifies that the row permission is to be enabled or disabled for row access control.

DISABLE
Specifies that the row permission is to be disabled for row access control. The row permission will remain ineffective regardless the row access control is activated for the table or not.

DISABLE is the default.

ENABLE
Specifies that the row permission is to be enabled for row access control. If row access control is not currently activated for the table, the row permission will become effective when row access control is activated for the table. If row access control is currently activated for the table, the row permission becomes effective immediately and all packages and dynamic cached statements that reference the table are invalidated.

Notes

How row permission are applied and how they affect certain statements:
See the ALTER TABLE statement with the ACTIVATE ROW ACCESS CONTROL clause for information on how to activate row access control and how row permissions are applied. See the description of subselect for information on how the application of row permissions affects the fetch operation. See the data change statements for information on how the application of row permissions affects the data change operation.

Row permissions that are created before row access control is activated for a table: The CREATE PERMISSION statement is an independent statement that can be used to create a row permission before row access control is activated for a table. The only requirement is that the table and the columns exist before the permission is created. Multiple row permissions can be created for a table.

The definition of the row permission is stored in the DB2 catalog. Dependency on the table for which the permission is being created and dependencies on other objects referenced in the definition are recorded. No package or dynamic cached statement is invalidated. A row permission can be created as enabled or disabled for row access control. An enabled row

permission does not take effect until the ALTER TABLE statement with the ACTIVATE ROW ACCESS CONTROL clause is used to activate row access control for the table. A disabled row permission remains ineffective even when row access control is activated for the table. The ALTER PERMISSION statement can be used to alter between ENABLE and DISABLE.

After row access control is activated for a table, when the table is referenced in a data manipulation statement, all enabled row permissions that are defined for the table are implicitly applied by DB2 to control access to the table.

Creating row permissions before activating row access control for a table is the recommended sequence to avoid multiple invalidations of packages and dynamic cached statements that reference the table.

Row permissions that are created after row access control is activated for a table: An enabled row permission becomes effective as soon as it is committed. All the packages and dynamic cached statements that reference the table are invalidated. Thereafter, when the table is referenced in a data manipulation statement, all enabled row permissions are implicitly applied to the statement. Any disabled row permission remains ineffective even when row access control is activated for the table.

No cascaded effect when row or column access control enforced tables are referenced in row permission definitions:
A row permission definition may reference tables and columns that are currently enforced by row or column access control. Access control from those tables are ignored when the table for which the row permission is being created is referenced in a data manipulation statement.

Multiple column masks and row permissions sharing the same environment variables:
Multiple column masks and row permissions can be created for a table. They must use the same set of environment variables. The set of environment variables is determined when the first column mask or row permission is created for the table.

The catalog table SYSENVIRONMENT contains the list of environment variables. The following table shows which environment variable must be the same among the multiple column masks and row permissions.

Table 116. Environment Variables in SYSIBM.SYSENVIRONMENT

Environment variables shown as SYSENVIRONMENT columns	Description	Static create statement	Dynamic create statement	Must be the same among multiple column masks and row permissions?
ENVID	Internal identifier of the environment	Assigned by DB2	Assigned by DB2	Yes
CURRENT_SCHEMA	The qualifier used to qualify unqualified objects such as tables, views. etc.	Package owner	Value of CURRENT_SCHEMA special register	Yes

Table 116. Environment Variables in SYSIBM.SYSENVIRONMENT (continued)

Environment variables shown as SYSENVIRONMENT columns	Description	Static create statement	Dynamic create statement	Must be the same among multiple column masks and row permissions?
PATHSCHEMAS	The schema path used to qualify unqualified object such as user-defined functions and CAST functions for user-defined data types.	PATH bind option	Value of CURRENT_PATH special register	Yes
APPLICATION_ ENCODING_ CCSID	The CCSID of the application environment	ENCODING bind option	CURRENT APPLICATION ENCODING SCHEME special register	Yes
ORIGINAL_ ENCODING_ CCSID	The original CCSID of the statement text string	CCSID(n) pre-compiler option or EBCDIC CCSID on DSNTIPF installation panel	CCSID based on DEF ENCODING SCHEME on DSNTIPF installation panel	Yes
DECIMAL_POINT	The decimal point indicator	COMMA or PERIOD precompiler option or DECIMAL POINT IS on DSNTIPF installation panelv	DECIMAL POINT IS on DSNTIPF installation panel	Yes
MIN_DIVIDE_SCALE	The minimum divide scale	MINIMUM DIVIDE SCALE on DSNTIP4 installation panel	MINIMUM DIVIDE SCALE on DSNTIP4 installation panelv	Yesv
STRING_DELIMITER	The string delimiter that is used in COBOL string constants	APOST precompiler option or STRING DELIMITER on DSNTIPF installation panel	STRING DELIMITER on DSNTIPF installation panel	No
SQL_ STRING_ DELIMITER	The SQL string delimiter that is used in constants	APOSTSQL pre-compiler option or SQL STRING DELIMITER on DSNTIPF installation panel	SQL STRING DELIMITER on DSNTIPF installation panel	Yes
MIXED_DATA	Uses mixed DBCS data	MIXED DATA on DSNTIPF installation panel	MIXED DATA on DSNTIPF installation panel	Yes
DECIMAL_ ARITHMETIC	The rules that are to be used for CURRENT PRECISION and when both operands in a decimal operation have a precision of 15 or less.	DEC(15) or DEC(31) precompiler option or DECIMAL ARITHMETIC on DSNTIP4 installation panel	DECIMAL ARITHMETIC on DSNTIP4 installation panel	Yes

Table 116. Environment Variables in SYSIBM.SYSENVIRONMENT (continued)

Environment variables shown as SYSENVIRONMENT columns	Description	Static create statement	Dynamic create statement	Must be the same among multiple column masks and row permissions?
DATE_FORMAT	The date format	DATE pre-compiler option or DATE FORMAT on DSNTIP4 installation panel	DATE FORMAT on DSNTIP4 installation panel	Yes
TIME_FORMAT	The time format	TIME pre-compiler option or TIME FORMAT on DSNTIP4 installation panel	TIME FORMAT on DSNTIP4 installation panel	Yes
FLOAT_FORMAT	The floating point format	FLOAT (S390 | IEEE) pre-compiler option or default of FLOAT S390	Default of FLOAT S390	No
HOST_LANGUAGE	The host language	HOST pre-compiler option or LANGUAGE DEFAULT on DSNTIPF installation panel	LANGUAGE DEFAULT on DSNTIPF installation panel	No
CHARSET	The character set	CCSID(n) pre-compiler option or EBCDIC CCSID on DSNTIPF installation panel	EBCDIC CCSID on DSNTIPF installation panel	No
FOLD	FOLD is only applicable when HOST_LANGUAGE is C or CPP. Otherwise FOLD is blank.	HOST(C(FOL D) precompiler option or default of NO FOLD	default of NO FOLD	No
ROUNDING	The rounding mode that is used when arithmetic and casting operations are performed on DECFLOAT data.	ROUNDING bind option	CURRENT DECFLOAT ROUNDING MODE special register	Yes

Note: In a data sharing environment, if a separate DSNHDECP module is provided for each member of the group, the DSNHDECP settings for each environment variable should be the same in all members of the data sharing group, otherwise an error might be issued when multiple column masks or row permissions are created.

Ordinary SQL identifiers specified in a static CREATE PERMISSION statement in a COBOL application:

If the CREATE PERMISSION statement is a static statement in a COBOL application, the ordinary SQL identifiers used in the row permission definition must not follow the rules for naming COBOL words (DSNH20474, reason code 14). They must follow the rules for naming SQL identifiers as described in the topic “SQL identifiers” in DB2 SQL Reference. For example, the COBOL word 1ST-TIME is not allowed as an

ordinary SQL identifier in a row permission definition; change it to FIRST_TIME or put it in the delimiters.

Encoding scheme and CCSIDs of the data manipulation statement after row permissions are applied:

The encoding scheme and CCSIDs of the data manipulation statement is not affected by the row permissions that are implicitly applied by DB2 for the row access control. The row permission definition is evaluated using its target table's encoding scheme and CCSIDs.

Consideration for DB2 limits:

If the data manipulation statement already approaches some DB2 limits in the statement, it should be noted that the more enabled row permissions and enabled column masks are created, the more likely they would impact some limits. For example, they may cause the statement to exceed the maximum total length (32600 bytes) of columns of a query operation requiring sort and evaluating aggregate functions (MULTIPLE DISTINCT and GROUP BY). This is because the enabled column mask and enabled row permission definitions are implicitly merged into the statement when the table is referenced in a data manipulation statement. See "Limits in DB2 for z/OS" in SQL Reference for the limits of a statement.

Restrictions involving pending definition changes:

CREATE PERMISSION is not allowed if the permission is defined on a table or references a table that has pending definition changes.

Examples

Example 1:

Secure user-defined function ACCOUNTING_UDF in row permission SALARY_ROW_ACCESS processes the sensitive data in column SALARY. After row access control is activated for table EMPLOYEE, Accountant Paul retrieves the salary of employee with EMPNO 123456 who is making $100,000 a year. Paul may or may not see the row depending on the output value from user-defined function ACCOUNTING_UDF.

```
CREATE PERMISSION SALARY_ROW_ACCESS ON EMPLOYEE
   FOR ROWS WHERE VERIFY_GROUP_FOR_USER(SESSION_USER,'MGR','ACCOUNTING') = 1
            AND
            ACCOUNTING_UDF(SALARY) < 120000
   ENFORCED FOR ALL ACCESS
   ENABLE;

COMMIT;

ALTER TABLE EMPLOYEE
 ACTIVATE ROW ACCESS CONTROL;

COMMIT;

SELECT SALARY FROM EMPLOYEE
   WHERE EMPNO = 123456;
```

Example 2:

The tellers in a bank can only access customers from their branch. All tellers have secondary authorization ID TELLER. The customer service representatives are allowed to access all customers of the bank. All customer service representatives have secondary authorization ID CSR. A row permission is created for each group of personnel in the bank accordingly to the access rule defined by SECADM authority. After row access control is activated for table CUSTOMER, in the SELECT statement

the search conditions of both row permissions are merged into the statement and they are combined with the logic OR operator to control the set of rows accessible by each group.

```
CREATE PERMISSION TELLER_ROW_ACCESS ON CUSTOMER
   FOR ROWS WHERE VERIFY_GROUP_FOR_USER(SESSION_USER,'TELLER') = 1
            AND
            BRANCH = (SELECT HOME_BRANCH FROM INTERNAL_INFO
                         WHERE EMP_ID = SESSION_USER)
    ENFORCED FOR ALL ACCESS
    ENABLE;

COMMIT;

CREATE PERMISSION CSR_ROW_ACCESS ON CUSTOMER
   FOR ROWS WHERE VERIFY_GROUP_FOR_USER(SESSION_USER,'CSR') = 1
   ENFORCED FOR ALL ACCESS
   ENABLE;

COMMIT;

ALTER TABLE CUSTOMER
   ACTIVATE ROW ACCESS CONTROL;

COMMIT;

SELECT * FROM CUSTOMER;
```

CREATE PROCEDURE

The CREATE PROCEDURE statement registers a stored procedure with a database server. You can register the following types of procedures with this statement, each of which is described separately.

- External

  The procedure is written in a programming language such as C, COBOL, or Java. The external executable is referenced by a procedure defined at the server along with various attributes of the procedure. See "CREATE PROCEDURE (external)" on page 1319.
- SQL (external)

  The procedure is written exclusively in SQL statements, but is implemented as an external program. The body of an SQL procedure is written in the SQL procedural language. The procedure body is defined at the current server along with various attributes of the procedure. See "CREATE PROCEDURE (SQL - external)" on page 1338.
- SQL (native)

  The procedure is written exclusively in SQL statements, but is implemented natively in DB2. The body of an SQL procedure is written in the SQL procedural language. The procedure body is defined at the current server along with various attributes of the procedure. See "CREATE PROCEDURE (SQL - native)" on page 1350.

CREATE PROCEDURE (external)

The CREATE PROCEDURE statement defines an external stored procedure at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is specified implicitly or explicitly.

Authorization

The privilege set that is defined below must include at least one of the following:

- The CREATEIN privilege on the schema
- SYSADM or SYSCTRL authority
- System DBADM

The authorization ID that matches the schema name implicitly has the CREATEIN privilege on the schema.

If the authorization ID that is used to create the procedure has installation SYSADM authority, the procedure is identified as system-defined procedure.

When LANGUAGE is JAVA and a *jar-name* is specified in the EXTERNAL NAME clause, the privilege set must include USAGE on the JAR file, the Java archive file.

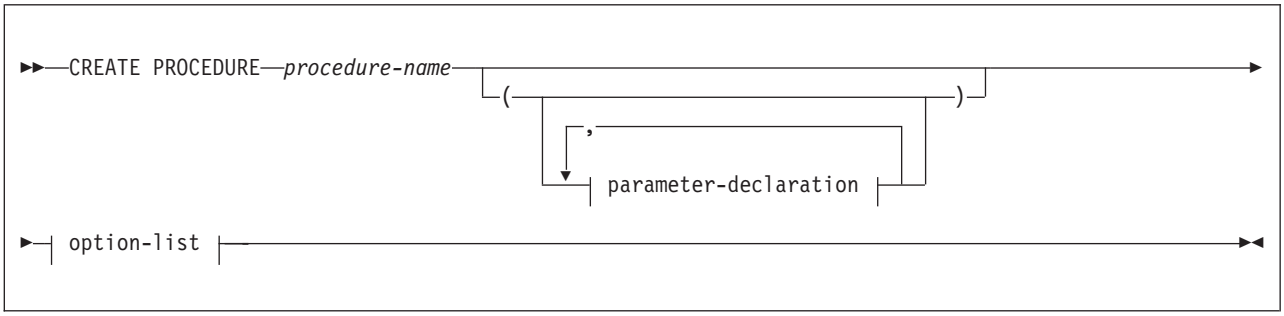
Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the owner is a role, the implicit schema match does not apply and this role needs to include one of the previously listed conditions.

If the statement is dynamically prepared and is not running in a trusted context for which the ROLE AS OBJECT OWNER clause is specified, the privilege set is the set of privileges that are held by the SQL authorization ID of the process. If the schema name is not the same as the SQL authorization ID of the process, one of the following conditions must be met:

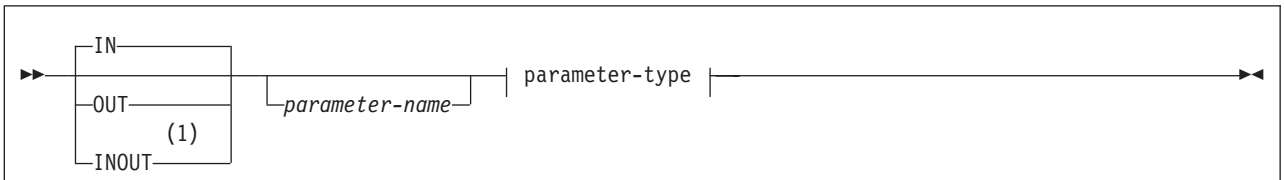
- The privilege set includes SYSADM or SYSCTRL authority.
- The SQL authorization ID of the process has the CREATEIN privilege on the schema.

The authorization ID that is used to create the stored procedure must have authority to define programs that run in the specified WLM environment. In addition, if the stored procedure uses a user-defined type as a parameter, this authorization ID must have the USAGE privilege on each parameter that is defined as a user-defined type.

Syntax



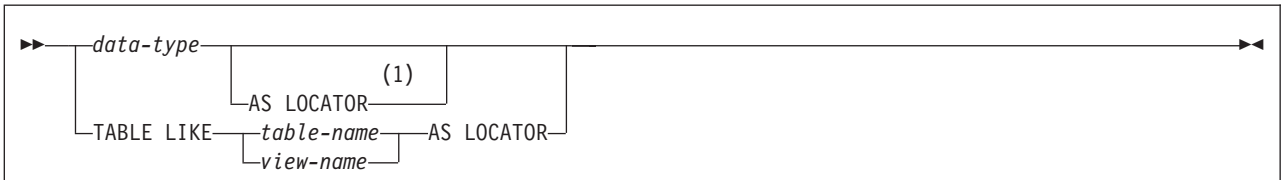
parameter-declaration:



Notes:

1 For a REXX stored procedure, only one parameter can have type OUT or INOUT. That parameter must be declared last.

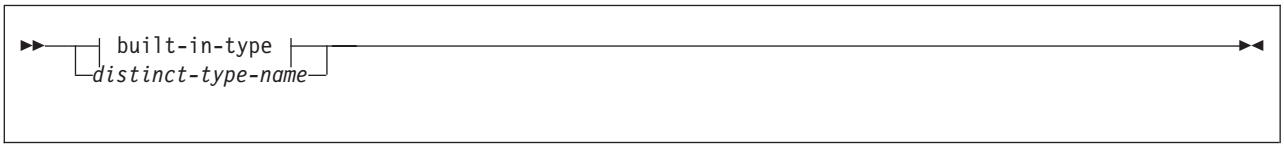
parameter-type:



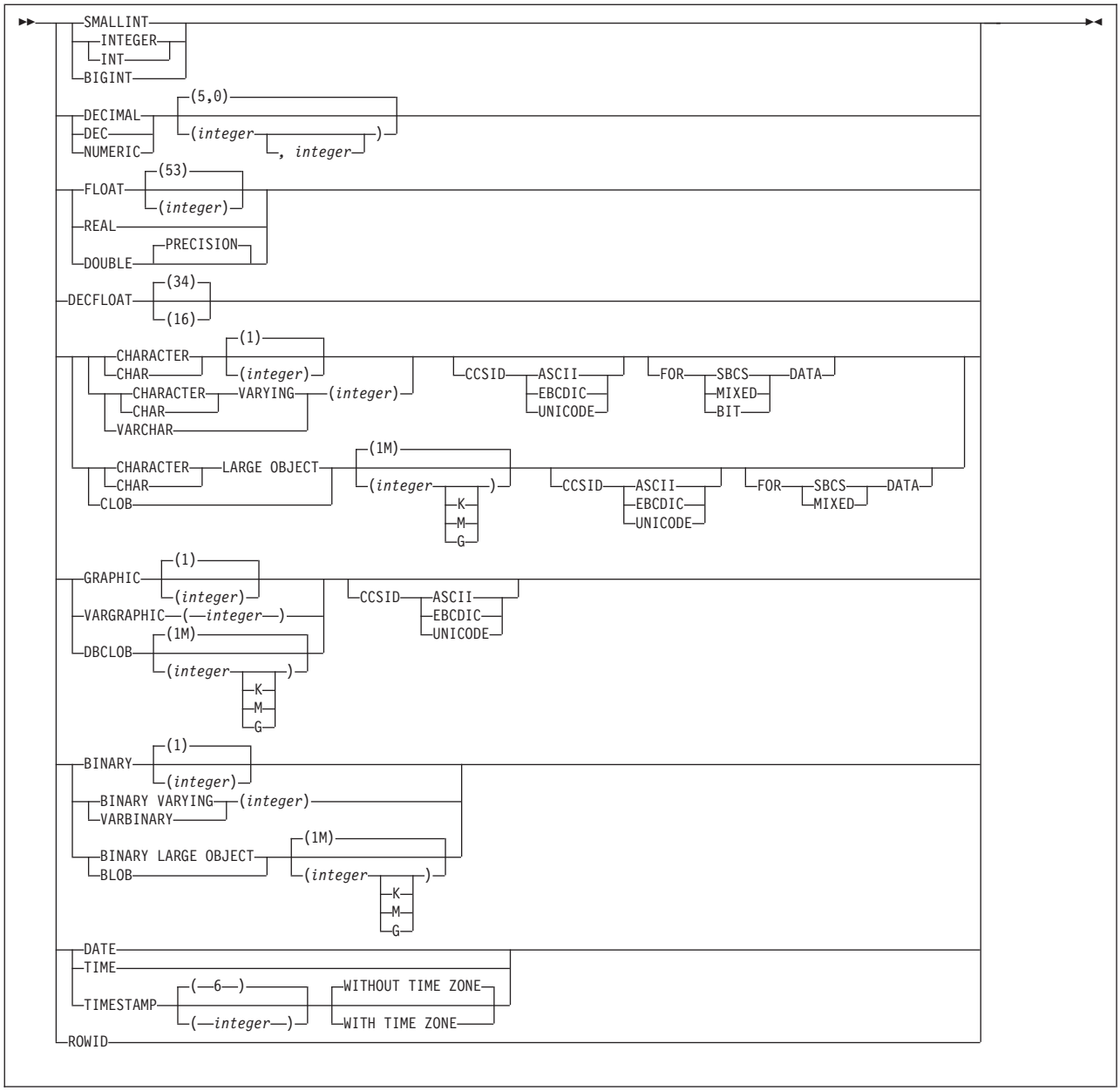
Notes:

1 AS LOCATOR can be specified only for a LOB data type or a distinct type based on a LOB data type.

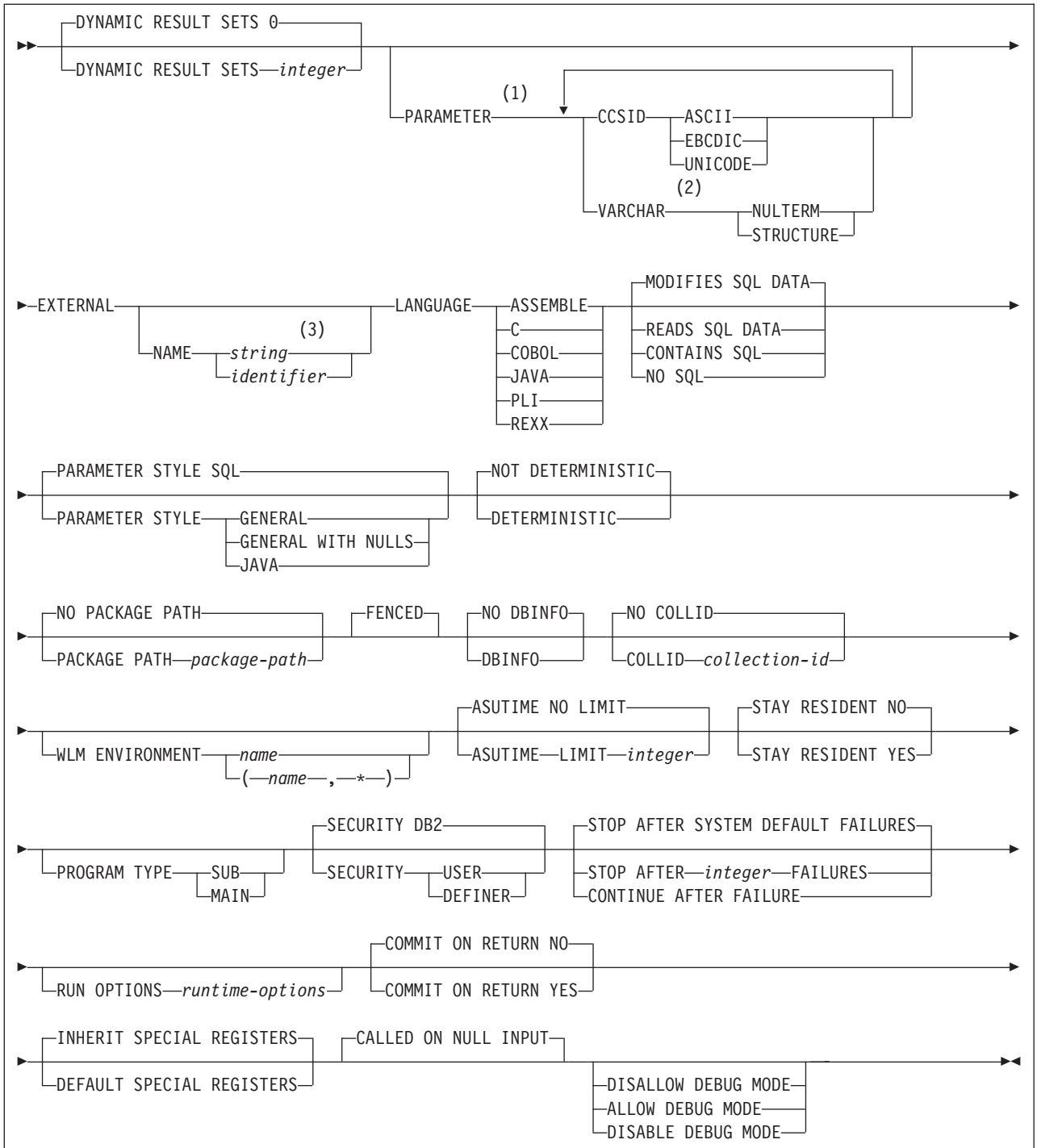
data-type:



built-in-type:



option-list: (The options can be specified in any order.)



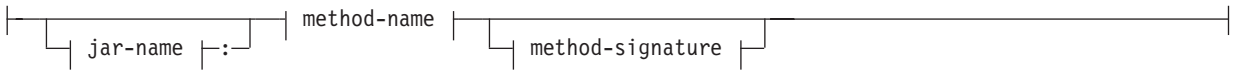
Notes:

1 The same clause must not be specified more than one time.

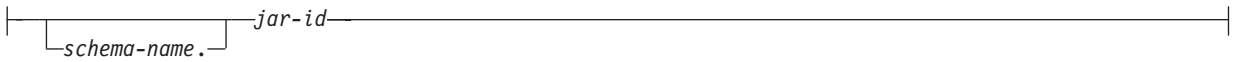
2 The VARCHAR clause can only be specified is LANGUAGE C is specified.

3 With LANGUAGE JAVA, use a valid *external-java-routine-name*.

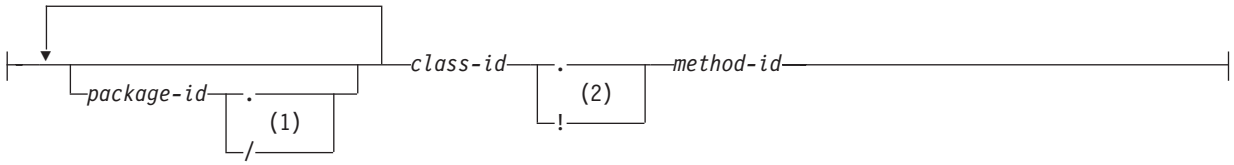
external-java-routine-name:



jar-name:



method-name:



method-signature:

Notes:

1 The slash (/) is supported for compatibility with previous releases of DB2 for z/OS.

2 The exclamation point (!) is supported for compatibility with other products in the DB2 family.

Description

procedure-name

Names the stored procedure. The name cannot be a single asterisk, even if you specify it as a delimited identifier ("*"). The name, including the implicit or explicit qualifier, must not identify an existing stored procedure at the current server.

The schema name can be 'SYSIBM' or 'SYSPROC'. It can also be 'SYSTOOLS' if the user who executes the CREATE statement has SYSADM or SYSCTRL privilege. Otherwise, the schema name must not begin with 'SYS' unless the schema name is 'SYSADM'.

(parameter-declaration,...)

Specifies the number of parameters of the stored procedure and the data type of each parameter, and optionally, the name of each parameter. A parameter for a stored procedure can be used only for input, only for output, or for both input and output. If an error is returned by the procedure, OUT parameters are undefined and INOUT parameters are unchanged.

All parameters are nullable except for numeric parameters in Java procedures, where numeric parameters, other than the DECIMAL types are not nullable in order to conform to the SQL/JRT standard.

IN Identifies the parameter as an input parameter to the procedure. The value of the parameter on entry to the procedure is the value that is returned to the calling SQL application, even if changes are made to the parameter within the procedure.

IN is the default.

OUT
Identifies the parameter as an output parameter that is returned by the stored procedure.

INOUT
Identifies the parameter as both an input and output parameter for the stored procedure.

parameter-name
Names the parameter for use as an SQL variable. The name cannot be the same as any other *parameter-name* for the procedure.

data-type
Specifies the data type of the parameter. The data type can be a built-in data type or a user-defined type.

If you specify the name of a user-defined type without a schema name, DB2 resolves the user-defined type by searching the schemas in the SQL path.

built-in-type
The data type of the parameter is a built-in data type.

For more information on the data types, see built-in-type.

For parameters with a character or graphic data type, the PARAMETER CCSID clause or CCSID clause indicates the encoding scheme of the parameter. If you do not specify either of these clauses, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

distinct-type-name
The data type of the input parameter is a distinct type. Any length, precision, scale, subtype, or encoding scheme attributes for the parameter are those of the source type of the distinct type.

Although an input parameter with a character data type has an implicitly or explicitly specified subtype (BIT, SBCS, or MIXED), the value that is actually passed in the input argument on the CALL statement can have any subtype. Therefore, conversion of the input data to the subtype of the parameter might occur when the procedure is called. With ASCII or EBCDIC, an error occurs if mixed data that actually contains DBCS characters is used as the value for an input parameter that is declared with an SBCS subtype.

Parameters with a datetime data type or a distinct type are passed to the function as a different data type:

- A datetime type parameter is passed as a character data type, and the data is passed in ISO format.

The encoding scheme for a datetime type parameter is the same as the implicitly or explicitly specified encoding scheme of any character or graphic string parameters. If no character or graphic string parameters are passed, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

- A distinct type parameter is passed as the source type of the distinct type.

AS LOCATOR
Specifies that a locator to the value of the parameter is passed to the procedure instead of the actual value. Specify AS LOCATOR only for parameters with a LOB data type or a distinct type based on a LOB data type. Passing locators instead of values can result in fewer bytes being passed to the procedure, especially when the value of the parameter is very large.

The AS LOCATOR clause has no effect on determining whether data types can be promoted.

TABLE LIKE *table-name* **or** *view-name* **AS LOCATOR**
Specifies that the parameter is a transition table. However, when the procedure is called, the actual values in the transition table are not passed to the stored procedure. A single value is passed instead. This single value is a locator to the table, which the procedure uses to access the columns of the transition table. A procedure with a table parameter can only be invoked from the triggered action of a trigger.

The use of TABLE LIKE provides an implicit definition of the transition table. It specifies that the transition table has the same number of columns as the identified table or view. If a table is specified, the transition table includes columns that are defined as implicitly hidden in the table. The columns have the same data type, length, precision, scale, subtype, and encoding scheme as the identified table or view, as they are described in catalog tables SYSCOLUMNS and SYSTABLESPACES. The number of columns and the attributes of those columns are determined at the time the CREATE PROCEDURE statement is processed. Any subsequent changes to the number of columns in the table or the attributes of those columns do not affect the parameters of the procedure.

table-name or *view-name* must identify a table or view that exists at the current server. The name must not identify a declared temporary table. The table that is identified can contain XML columns; however, the procedure cannot reference those XML columns. The name does not have to be the same name as the table that is associated with the transition table for the trigger. An unqualified table or view name is implicitly qualified according to the following rules:

- If the CREATE PROCEDURE statement is embedded in a program, the implicit qualifier is the authorization ID in the QUALIFIER bind option when the plan or package was created or last rebound. If QUALIFIER was not used, the implicit qualifier is the owner of the plan or package.
- If the CREATE PROCEDURE statement is dynamically prepared, the implicit qualifier is the SQL authorization ID in the CURRENT SCHEMA special register.

When the procedure is called, the corresponding columns of the transition table identified by the table locator and the table or view identified in the TABLE LIKE clause must have the same definition. The data type, length,

precision, scale, and encoding scheme of these columns must match exactly. The description of the table or view at the time the CREATE PROCEDURE statement was executed is used.

Additionally, a character FOR BIT DATA column of the transition table cannot be passed as input for a table parameter for which the corresponding column of the table specified at the definition is not defined as character FOR BIT DATA. (The definition occurs with the CREATE PROCEDURE statement.) Likewise, a character column of the transition table that is not FOR BIT DATA cannot be passed as input for a table parameter for which the corresponding column of the table specified at the definition is defined as character FOR BIT DATA.

For more information about using table locators, see *DB2 Application Programming and SQL Guide*.

FENCED
Specifies that the procedure runs in an external address space.

DYNAMIC RESULT SETS *integer*
Specifies the maximum number of query result sets that the stored procedure can return. The default is DYNAMIC RESULT SETS 0, which indicates that there are no result sets. The value must be between 0 and 32767.

ALLOW DEBUG MODE, DISALLOW DEBUG MODE, or DISABLE DEBUG MODE
Specifies whether the procedure can be run in debugging mode. When DYNAMICRULES run behavior is in effect, the default is determined by using the value of the CURRENT DEBUG MODE special register. Otherwise the default is DISALLOW DEBUG MODE.

Do not specify this option unless LANGUAGE JAVA is in effect.

ALLOW DEBUG MODE
Specifies that the JAVA procedure can be run in debugging mode.

DISALLOW DEBUG MODE
Specifies that the JAVA procedure cannot be run in debugging mode.

You can use an ALTER PROCEDURE statement to change this option to ALLOW DEBUG MODE.

DISABLE DEBUG MODE
Specifies that the JAVA procedure can never be run in debugging mode.

The procedure cannot be changed to specify ALLOW DEBUG MODE or DISALLOW DEBUG MODE once the procedure has been created or altered using DISABLE DEBUG MODE. To change this option, you must drop and re-create the procedure using the option that you want.

PARAMETER CCSID or PARAMETER VARCHAR
Specifies the encoding scheme for string parameters, and in the case of LANGUAGE C, specifies the representation of variable length string parameters.

CCSID
Indicates whether the encoding scheme for character or graphic string parameters is ASCII, EBCDIC, or UNICODE. The default encoding scheme is the value specified in the CCSID clauses of the parameter list or in the field DEF ENCODING SCHEME on installation panel DSNTIPF.

This clause provides a convenient way to specify the encoding scheme for character or graphic string parameters. If individual CCSID clauses are specified for individual parameters in addition to this PARAMETER

CCSID clause, the value specified in *all* of the CCSID clauses must be the same value that is specified in this clause.

This clause also specifies the encoding scheme to be used for system-generated parameters of the routine such as message tokens and DBINFO.

VARCHAR
Specifies that the representation of the values of varying length character string-parameters for procedures that specify LANGUAGE C.

This option can only be specified if LANGUAGE C is also specified.

NULTERM
Specifies that variable length character string parameters are represented in a NUL-terminated string form.

STRUCTURE
Specifies that variable length character string parameters are represented in a VARCHAR structure form.

Using the PARAMETER VARCHAR clause, there is no way to specify the VARCHAR form of an individual parameter as these is with PARAMETER CCSID. The PARAMETER VARCHAR clause only applies to parameters in the parameter list of a procedure and in the RETURNS clause. It does not apply to system-generated parameters of the routine such as message tokens and DBINFO.

In a data sharing environment, you should not specify the PARAMETER VARCHAR clause until all members of the data sharing group support the clause. If some group members support this clause and others do not, and PARAMETER VARCHAR is specified in an external routine, the routine will encounter different parameter forms depending on which group member invokes the routine.

EXTERNAL
Specifies that the CREATE PROCEDURE statement is being used to define a new procedure that is based on code written in an external programming language. If the NAME clause is not specified, 'NAME *procedure-name*' is assumed. The NAME clause is required for a LANGUAGE JAVA procedure because the default name is not valid for a Java procedure. In some cases, the default name will not be valid. To avoid invalid names, specify the NAME clause for the following types of procedures:

- A procedure that is defined as LANGUAGE JAVA
- A procedure that has a name that is greater than 8 bytes in length, contains an underscore, or does not conform to the rules for an ordinary identifier.

NAME *string* **or** *identifier*
Identifies the user-written code that implements the stored procedure.

If LANGUAGE is JAVA, *string* must be specified and enclosed in single quotation marks, with no extraneous blanks within the single quotation marks. It must specify a valid *external-java-routine-name*. If multiple *string*s are specified, the total length of all of them must not be greater than 1305 bytes and they must be separated by a space or a line break.

An *external-java-routine-name* contains the following parts:

jar-name
Identifies the name given to the JAR file when it was installed in the database. The name contains *jar-id*, which can optionally be qualified

with a schema. Examples are "myJar" and "mySchema.myJar." The unqualified *jar-id* is implicitly qualified with a schema name according to the following rules:

- If the statement is embedded in a program, the schema name is the authorization ID in the QUALIFIER bind option when the package or plan was created or last rebound. If the QUALIFIER was not specified, the schema name is the owner of the package or plan.
- If the statement is dynamically prepared, the schema name is the SQL authorization ID in the CURRENT SCHEMA special register.

If *jar-name* is specified, it must exist when the CREATE PROCEDURE statement is processed. Do not specify a *jar-name* for a JAVA procedure for which NO SQL is also specified.

If *jar-name* is not specified, the procedure is loaded from the class file directly instead of being loaded from a JAR file. DB2 searches the directories in the CLASSPATH associated with the WLM Environment. Environmental variables for Java routines are specified in a data set identified in a JAVAENV DD card on the JCL used to start the address space for a WLM-managed stored procedure.

method-name
Identifies the name of the method and must not be longer than 254 bytes. Its package, class, and method ID's are specific to Java and as such are not limited to 18 bytes. In addition, the rules for what these can contain are not necessarily the same as the rules for an SQL ordinary identifier.

package-id
Identifies a package. The concatenated list of *package-ids* identifies the package that the class identifier is part of. If the class is part of a package, the method name must include the complete package prefix, such as "myPacks.StoredProcs." The Java virtual machine looks in the directory "/myPacks/StoredProcs/" for the classes.

class-id
Identifies the class identifier of the Java object.

method-id
Identifies the method identifier with the Java class to be invoked.

method-signature
Identifies a list of zero or more Java data types for the parameter list and must not be longer than 1024 bytes. Specify the *method-signature* if the procedure involves any input or output parameters that can be NULL. When the stored procedure being created is called, DB2 searches for a Java method with the exact *method-signature*. The number of *java-datatype* elements specified indicates how many parameters that the Java method must have.

A Java procedure can have no parameters. In this case, you code an empty set of parentheses for *method-signature*. If a Java *method-signature* is not specified, DB2 searches for a Java method with a signature derived from the default JDBC types associated with the SQL types specified in the parameter list of the CREATE PROCEDURE statement.

For other values of LANGUAGE, the value must conform to the naming conventions for MVS load modules: the value must be less than or equal to 8 bytes, and it must conform to the rules for an ordinary identifier with the exception that it must not contain an underscore.

LANGUAGE
This mandatory clause is used to specify the language interface convention to which the procedure body is written. All programs must be designed to run in the server's environment. Assembler, C, COBOL, and PL/I programs must be designed to run in IBM's Language Environment.

ASSEMBLE
The stored procedure is written in Assembler.

C The stored procedure is written in C or C++.

COBOL
The stored procedure is written in COBOL, including the OO-COBOL language extensions.

JAVA
The stored procedure is written in Java and is executed in the Java Virtual Machine. When LANGUAGE JAVA is specified, the EXTERNAL NAME clause must be specified with a valid *external-java-routine-name* and PARAMETER STYLE must be specified with JAVA. The procedure must be a public static method of the specified Java class.

Do not specify LANGUAGE JAVA when DBINFO, PROGRAM TYPE MAIN, or RUN OPTIONS is specified.

PLI
The stored procedure is written in PL/I.

REXX
The stored procedure is written in REXX. Do not specify LANGUAGE REXX when PARAMETER STYLE SQL is in effect. When REXX is specified, the procedure must use PARAMETER STYLE GENERAL or GENERAL WITH NULLS.

MODIFIES SQL DATA, READS SQL DATA, CONTAINS SQL, or NO SQL
Specifies which SQL statements, if any, can be executed in the procedure or any routine that is called from this procedure. The default is MODIFIES SQL DATA. For the data access classification of each statement, see Table 162 on page 2030.

MODIFIES SQL DATA
Specifies that the procedure can execute any SQL statement except statements that are not supported in procedures.

READS SQL DATA
Specifies that the procedure can execute statements with a data access indication of READS SQL DATA, CONTAINS SQL, or NO SQL. The procedure cannot execute SQL statements that modify data.

CONTAINS SQL
Specifies that the procedure can execute only SQL statements with an access indication of CONTAINS SQL or NO SQL. The procedure cannot execute statements that read or modify data.

NO SQL
Specifies that the procedure can execute only SQL statements with a data access classification of NO SQL. Do not specify NO SQL for a JAVA procedure that uses a JAR file.

PARAMETER STYLE
Identifies the linkage convention used to pass parameters to and return values from the stored procedure. All of the linkage conventions provide arguments to

the stored procedure that contain the parameters specified on the CALL statement. Some of the linkage conventions pass additional arguments to the stored procedure that provide more information to the stored procedure. For more information on linkage conventions, see *DB2 Application Programming and SQL Guide*.

SQL
Specifies that, in addition to the parameters on the CALL statement, several additional parameters are passed to the stored procedure. The following parameters are passed:

- The first *n* parameters that are specified on the CREATE PROCEDURE statement.
- *n* parameters for indicator variables for the parameters.
- The SQLSTATE to be returned.
- The qualified name of the stored procedure.
- The specific name of the stored procedure.
- The SQL diagnostic string to be returned to DB2.
- If DBINFO is specified, the DBINFO structure.

PARAMETER STYLE SQL is the default. Do not specify PARAMETER STYLE SQL when LANGUAGE REXX or LANGUAGE JAVA is in effect.

GENERAL
Specifies that the stored procedure uses a parameter passing mechanism where the stored procedure receives only the parameters specified on the CALL statement. Arguments to procedures defined with this parameter style cannot be null.

GENERAL WITH NULLS
Specifies that, in addition to the parameters on the CALL statement as specified in GENERAL, another argument is also passed to the stored procedure. The additional argument contains an indicator array with an element for each of the parameters on the CALL statement. In C, this is an array of short INTS. The indicator array enables the stored procedure to accept or return null parameter values.

JAVA
Specifies that the stored procedure uses a parameter passing convention that conforms to the Java and SQLJ Routines specifications. PARAMETER JAVA can be specified only if LANGUAGE is JAVA. JAVA must be specified for PARAMETER STYLE when LANGUAGE is JAVA.

INOUT and OUT parameters are passed as single-entry arrays. The INOUT and OUT parameters are declared in the Java method as single-element arrays of the Java type.

For REXX stored procedures (LANGUAGE REXX), GENERAL and GENERAL WITH NULLS are the only valid values for PARAMETER STYLE; therefore, specify one of these values and do not allow PARAMETER STYLE to default to SQL.

DETERMINISTIC or NOT DETERMINISTIC
Specifies whether the stored procedure returns the same results each time the stored procedure is called with the same IN and INOUT arguments.

DETERMINISTIC
The stored procedure always returns the same results each time the stored

procedure is called with the same IN and INOUT arguments, if the referenced data in the database has not changed.

NOT DETERMINISTIC
The stored procedure might not return the same result each time the procedure is called with the same IN and INOUT arguments, even when the referenced data in the database has not changed. NOT DETERMINISTIC is the default.

DB2 does not verify that the stored procedure code is consistent with the specification of DETERMINISTIC or NOT DETERMINISTIC.

NO PACKAGE PATH or PACKAGE PATH *package-path*
Specifies the package path to use when the procedure is run. This is the list of the possible package collections into which the DBRM this is associated with the procedure is bound.

NO PACKAGE PATH
Specifies that the list of package collections for the procedure is the same as the list of package collection IDs for the calling program. If the calling program does not use a package, DB2 resolves the package by using the CURRENT PACKAGE PATH special register, the CURRENT PACKAGESET special register, or the PKLIST bind option (in this order). For information about how DB2 uses these three items, see *DB2 Application Programming and SQL Guide*.

PACKAGE PATH *package-path*
Specifies a list of package collections, in the same format as the SET CURRENT PACKAGE PATH special register.

If the COLLID clause is specified with PACKAGE PATH, the COLLID clause is ignored when the routine is invoked.

The *package-path* value that is provided when the procedure is created is checked when the CALL statement is prepared. If *package-path* contains SESSION_USER (or USER), PATH, or PACKAGE PATH, an error is returned when the *package-path* value is checked.

NO DBINFO or DBINFO
Specifies whether additional status information is passed to the stored procedure when it is invoked.

NO DBINFO
Additional information is not passed. NO DBINFO is the default.

DBINFO
An additional argument is passed when the stored procedure is invoked. The argument is a structure that contains information such as the name of the current server, the application run time authorization ID and identification of the version and release of the database manager that invoked the procedure. For details about the argument and its structure, see *DB2 Application Programming and SQL Guide*.

DBINFO can be specified only if PARAMETER STYLE SQL is specified.

NO COLLID or COLLID *collection-id*
Identifies the package collection that is to be used when the stored procedure is executed. This is the package collection into which the DBRM that is associated with the stored procedure is bound.

NO COLLID
The package collection for the stored procedure is the same as the package

collection of the calling program. If the invoking program does not use a package, DB2 resolves the package by using the CURRENT PACKAGE PATH special register, the CURRENT PACKAGESET special register, or the PKLIST bind option (in this order). For details about how DB2 uses these three items, see the information on package resolution in *DB2 Application Programming and SQL Guide*.

NO COLLID is the default.

COLLID *collection-id*
The package collection for the stored procedure is the one specified.

For REXX stored procedures, *collection-id* can be DSNREXRR, DSNREXRS, DSNREXCR, or DSNREXCS.

WLM ENVIRONMENT
Identifies the WLM (workload manager) environment in which the stored procedure is to run when the DB2 stored procedure address space is WLM-established. The *name* of the WLM environment is an SQL identifier.

If you do not specify WLM ENVIRONMENT, the stored procedure runs in the default WLM-established stored procedure address space specified at installation time.

name
The WLM environment in which the stored procedure must run. If another stored procedure or a user-defined function calls the stored procedure and that calling routine is running in an address space that is not associated with the specified WLM environment, DB2 routes the stored procedure request to a different address space.

(name,*)
When an SQL application program directly calls a stored procedure, the WLM environment in which the stored procedure runs.

If another stored procedure or a user-defined function calls the stored procedure, the stored procedure runs in the same WLM environment that the calling routine uses.

To define a stored procedure that is to run in a specified WLM environment, you must have appropriate authority for the WLM environment. For an example of a RACF command that provides this authorization, see Running stored procedures.

ASUTIME
Specifies the total amount of processor time, in CPU service units, that a single invocation of a stored procedure can run. The value is unrelated to the ASUTIME column of the resource limit specification table. This option is ignored if LANGUAGE JAVA is specified.

When you are debugging a stored procedure, setting a limit can be helpful in case the stored procedure gets caught in a loop. For information on service units, see *z/OS MVS Initialization and Tuning Guide*.

NO LIMIT
There is no limit on the service units. NO LIMIT is the default.

LIMIT *integer*
The limit on the number of CPU service units is a positive *integer* in the range of 1 to 2 147 483 647. If the procedure uses more service units than

the specified value, DB2 cancels the procedure. The CPU cycles that are consumed by parallel tasks in a procedure do not contribute towards the specified ASUTIME LIMIT.

STAY RESIDENT
Specifies whether the stored procedure load module is to remain resident in memory when the stored procedure ends. This option is ignored if LANGUAGE JAVA is specified.

NO The load module is deleted from memory after the stored procedure ends. NO is the default.

YES
The load module remains resident in memory after the stored procedure ends.

PROGRAM TYPE
Specifies whether the stored procedure runs as a main routine or a subroutine.

SUB
The stored procedure runs as a subroutine. With LANGUAGE JAVA, PROGRAM TYPE SUB is the only valid option.

MAIN
The stored procedure runs as a main routine. With LANGUAGE REXX, PROGRAM TYPE MAIN is always in effect.

The default for PROGRAM TYPE is:

- MAIN with LANGUAGE REXX
- SUB with LANGUAGE JAVA
- For other languages, the default depends on the value of the CURRENT RULES special register:
  - MAIN when the value is DB2
  - SUB when the value is STD

SECURITY
Specifies how the stored procedure interacts with an external security product, such as RACF, to control access to non-SQL resources.

DB2
The stored procedure does not require a special external security environment. If the stored procedure accesses resources that an external security product protects, the access is performed using the authorization ID associated with the stored procedure address space. DB2 is the default.

USER
An external security environment should be established for the stored procedure. If the stored procedure accesses resources that the external security product protects, the access is performed using the authorization ID of the user who invoked the stored procedure.

DEFINER
An external security environment should be established for the stored procedure. If the stored procedure accesses resources that the external security product protects, the access is performed using the authorization ID of the owner of the stored procedure.

STOP AFTER SYSTEM DEFAULT FAILURES, STOP AFTER *nn* FAILURES, or CONTINUE AFTER FAILURE
Specifies whether the routine is to be put in a stopped state after some number of failures.

STOP AFTER SYSTEM DEFAULT FAILURES
Specifies that this routine should be placed in a stopped state after the number of failures indicated by the value of field MAX ABEND COUNT on installation panel DSNTIPX. This is the default.

STOP AFTER *nn* **FAILURES**
Specifies that this routine should be placed in a stopped state after *nn* failures. The value *nn* can be an integer from 1 to 32767.

CONTINUE AFTER FAILURE
Specifies that this routine should not be placed in a stopped state after any failure.

RUN OPTIONS *runtime-options*
Specifies the Language Environment run time options to be used for the stored procedure. For a REXX stored procedure, specifies the Language Environment run time options to be passed to the REXX language interface to DB2. You must specify *runtime-options* as a character string that is no longer than 254 bytes. If you do not specify RUN OPTIONS or pass an empty string, DB2 does not pass any run time options to Language Environment, and Language Environment uses its installation defaults.

Do not specify RUN OPTIONS when LANGUAGE JAVA is in effect.

For a description of the Language Environment run time options, see *z/OS Language Environment Programming Reference*.

COMMIT ON RETURN
Indicates whether DB2 commits the transaction immediately on return from the stored procedure.

NO DB2 does not issue a commit when the stored procedure returns. NO is the default.

YES
DB2 issues a commit when the stored procedure returns if the following statements are true:

- The SQLCODE that is returned by the CALL statement is not negative.
- The stored procedure is not in a must abort state.

The commit operation includes the work that is performed by the calling application process and the stored procedure.

If the stored procedure returns result sets, the cursors that are associated with the result sets must have been defined as WITH HOLD to be usable after the commit.

INHERIT SPECIAL REGISTERS or DEFAULT SPECIAL REGISTERS
Specifies how special registers are set on entry to the routine. The default is INHERIT SPECIAL REGISTERS.

INHERIT SPECIAL REGISTERS
Specifies that the values of special registers are inherited according to the rules listed in the table for characteristics of special registers in a stored procedure in Table 40 on page 205.

DEFAULT SPECIAL REGISTERS
Specifies that special registers are initialized to the default values, as indicated by the rules in the table for characteristics of special registers in a stored procedure in Table 40 on page 205.

CALLED ON NULL INPUT
Specifies that the procedure is to be called even if any or all argument values are null, which means that the procedure must be coded to test for null argument values. The procedure can return null or nonnull values. CALLED ON NULL INPUT is the default.

Notes

Owner privileges: The owner is authorized to call the procedure (EXECUTE privilege) and grant others the privilege to call the procedure. See "GRANT (function or procedure privileges)" on page 1703. For more information about ownership of the object, see "Authorization, privileges, permissions, masks, and object ownership" on page 70.

Choosing data types for parameters: When you choose the data types of the parameters for your stored procedure, consider the rules of promotion that can affect the values of the parameters. (See "Promotion of data types" on page 110). For example, a constant that is one of the input arguments to the stored procedure might have a built-in data type that is different from the data type that the procedure expects, and more significantly, might not be promotable to that expected data type. Based on the rules of promotion, using the following data types for parameters is recommended:

- INTEGER instead of SMALLINT
- DOUBLE instead of REAL
- VARCHAR instead of CHAR
- VARGRAPHIC instead of GRAPHIC
- VARBINARY instead of BINARY

For portability of functions across platforms that are not DB2 for z/OS, do not use the following data types, which might have different representations on different platforms:

- FLOAT. Use DOUBLE or REAL instead.
- NUMERIC. Use DECIMAL instead.

Specifying the encoding scheme for parameters: The encoding scheme of all of the parameters with a character or graphic string data type (both input and output parameters) must be the same—either all ASCII, all EBCDIC, or all UNICODE. If you specify the encoding scheme on the individual parameters, instead of using the PARAMETER CCSID to specify it for all parameters at once or allowing the encoding scheme to default to the system value, ensure that they all agree.

Character string representation considerations: The PARAMETER VARCHAR clause is specific to LANGUAGE C routines because of the native use of NUL-terminated strings in C. VARCHAR structure representation is useful when character string data is known to contain embedded NUL-terminators. It is also useful when it cannot be guaranteed that character string data does not contain embedded NUL-terminators.

PARAMETER VARCHAR does not apply to fixed length character strings, VARCHAR FOR BIT DATA, CLOB, DBCLOB, or implicitly generated parameters. The clause does not apply to VARCHAR FOR BIT DATA because BIT DATA can contain X'00' characters, and its value representation starts with length information. It does not apply to LOB data because a LOB value representation starts with length information.

PARAMETER VARCHAR does not apply to optional parameters that are implicitly provided to an external procedure. For example, a CREATE PROCEDURE statement for LANGUAGE C must also specify PARAMETER STYLE SQL, which returns an SQLSTATE NUL-terminated character string; that SQLSTATE will not be represented in VARCHAR structured form. Likewise, none of the parameters that represent the qualified name of the procedure, the specific name of the procedure, or the SQL diagnostic string that is returned to the database manager will be represented in VARCHAR structured form.

Running stored procedures: You can use the WLM ENVIRONMENT clause to identify the address space in which a stored procedure is to run. Using different WLM environments lets you isolate one group of programs from another. For example, you might choose to isolate programs based on security requirements and place all payroll applications in one WLM environment because those applications deal with sensitive data, such as employee salaries.

Regardless of where the stored procedure is to run, DB2 invokes RACF to determine whether you have appropriate authorization. You must have authorization to issue CREATE PROCEDURE statements that refer to the specified WLM environment or the DB2-established stored procedure address space. For example, the following RACF command authorizes DB2 user DB2USER1 to define stored procedures on DB2 subsystem DB2A that run in the WLM environment named PAYROLL.

```
PERMIT  DB2A.WLMENV.PAYROLL  CLASS(DSNR) ID(DB2USER1)  ACCESS(READ)
```

Accessing result sets from nested stored procedures: When another stored procedure or a user-defined function calls a stored procedure, only the calling routine can access the result sets that the stored procedure returns. The result sets are not returned to the application that contains the outermost stored procedure or user-defined function in the sequence of nested calls.

When a stored procedure is nested, the result sets that are returned by the stored procedure are accessible only by the calling routine. The result sets are not returned to the application that contains the outermost stored procedure or user-defined function in the sequence of nested calls.

Restrictions for nested stored procedures: A stored procedure, user-defined function, or trigger cannot call a stored procedure that is defined with the COMMIT ON RETURN clause.

Stored procedures and user-defined session global variables:

The content of user-defined session global variables that are referenced in routines is inherited from the caller. User-defined session global variables can be modified in stored procedures, except when the stored procedure is called by a trigger or a function.

If the procedure contains references to user-defined session global variables, the level of SQL data access must be at least CONTAINS SQL. If the procedure contains SQL statements that modify user-defined session global variables, the level of SQL data access must be MODIFIES SQL DATA.

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords:

- RESULT SET as a synonym for DYNAMIC RESULT SET
- RESULT SETS as a synonym for DYNAMIC RESULT SETS

- STANDARD CALL as a synonym for DB2SQL
- SIMPLE CALL as a synonym for GENERAL
- SIMPLE CALL WITH NULLS as a synonym for GENERAL WITH NULLS
- VARIANT as a synonym for NOT DETERMINISTIC
- NOT VARIANT as a synonym for DETERMINISTIC
- NULL CALL as a synonym for CALLED ON NULL INPUT
- PARAMETER STYLE DB2SQL as a synonym for PARAMETER STYLE SQL

Examples

Example 1: Create the definition for a stored procedure that is written in COBOL. The procedure accepts an assembly part number and returns the number of parts that make up the assembly, the total part cost, and a result set. The result set lists the part numbers, quantity, and unit cost of each part. Assume that the input parameter cannot contain a null value and that the procedure is to run in a WLM environment called PARTSA.

```
CREATE PROCEDURE SYSPROC.MYPROC(IN INT, OUT INT, OUT DECIMAL(7,2))
       LANGUAGE COBOL
       EXTERNAL NAME MYMODULE
       PARAMETER STYLE GENERAL
       WLM ENVIRONMENT PARTSA
       DYNAMIC RESULT SETS 1;
```

Example 2: Create the definition for the stored procedure described in Example 1, except use the linkage convention that passes more information than the parameter specified on the CALL statement. Specify Language Environment run time options HEAP, BELOW, ALL31, and STACK.

```
CREATE PROCEDURE SYSPROC.MYPROC(IN INT, OUT INT, OUT DECIMAL(7,2))
       LANGUAGE COBOL
       EXTERNAL NAME MYMODULE
       PARAMETER STYLE SQL
       WLM ENVIRONMENT PARTSA
       DYNAMIC RESULT SETS 1
       RUN OPTIONS 'HEAP(,,ANY),BELOW(4K,,),ALL31(ON),STACK(,,ANY,)';
```

Example 3: Create the procedure definition for a stored procedure, written in Java, that is passed a part number and returns the cost of the part and the quantity that is currently available.

```
CREATE PROCEDURE PARTS_ON_HAND(IN PARTNUM INT,
                               OUT COST DECIMAL(7,2),
                               OUT QUANTITY INT)
       LANGUAGE JAVA
       EXTERNAL NAME 'PARTS.ONHAND'
       PARAMETER STYLE JAVA;
```

CREATE PROCEDURE (SQL - external)

The CREATE PROCEDURE statement defines an external SQL procedure at the current server and specifies the source statements for the procedure. This is the only type of SQL procedure that is available for versions of DB2 prior to Version 9.

For information about the SQL control statements that are supported in external SQL procedures, refer to "SQL control statements for external SQL procedures" on page 2032.

Invocation

This statement can only be dynamically prepared, but the DYNAMICRULES run behavior must be specified implicitly or explicitly. It is intended to be processed using one of the following methods:

- JCL
- The DB2 for z/OS SQL procedure processor (DSNTPSMP) (IBM Optim™ Development Studio uses this method.)

Issuing the CREATE PROCEDURE statement from another context will result in an incomplete procedure definition even though the statement processing returns without error. For more information on preparing SQL procedures for execution, see *DB2 Application Programming and SQL Guide*.

Authorization

The privilege set that is defined below must include at least one of the following:

- The CREATEIN privilege on the schema
- SYSADM or SYSCTRL authority
- System DBADM

The authorization ID that matches the schema name implicitly has the CREATEIN privilege on the schema.

If the authorization ID that is used to create the procedure has installation SYSADM authority, the procedure is identified as system-defined procedure.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the owner is a role, the implicit schema match does not apply and this role needs to include one of the previously listed conditions.

If the statement is dynamically prepared and is not running in a trusted context for which the ROLE AS OBJECT OWNER clause is specified, the privilege set is the set of privileges that are held by the SQL authorization ID of the process. If the schema name is not the same as the SQL authorization ID of the process, one of the following conditions must be met:

- The privilege set includes SYSADM or SYSCTRL authority.
- The SQL authorization ID of the process has the CREATEIN privilege on the schema.

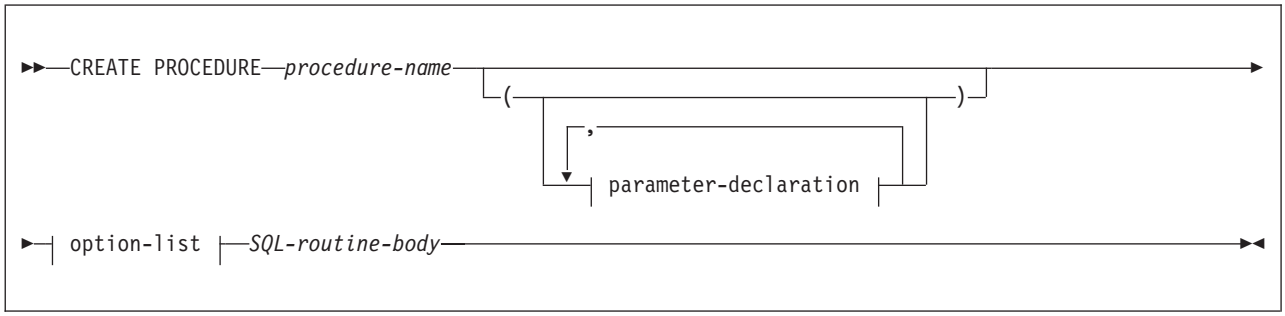
The authorization ID that is used to create the stored procedure must have authority to create programs that are to be run in the specified WLM environment.

The owner of the procedure is determined by how the CREATE PROCEDURE statement is invoked:

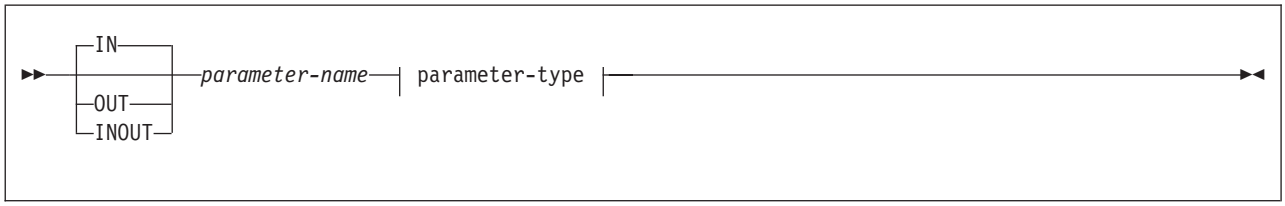
- If the statement is embedded in a program, the owner is the authorization ID of the owner of the plan or package.
- If the statement is dynamically prepared, the owner is the SQL authorization ID in the CURRENT SQLID special register.

The owner is implicitly given the EXECUTE privilege with the GRANT option for the procedure.

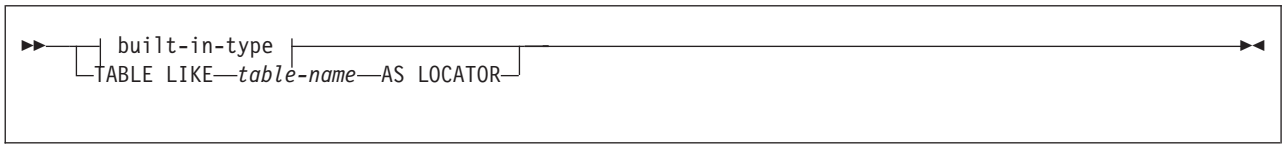
Syntax



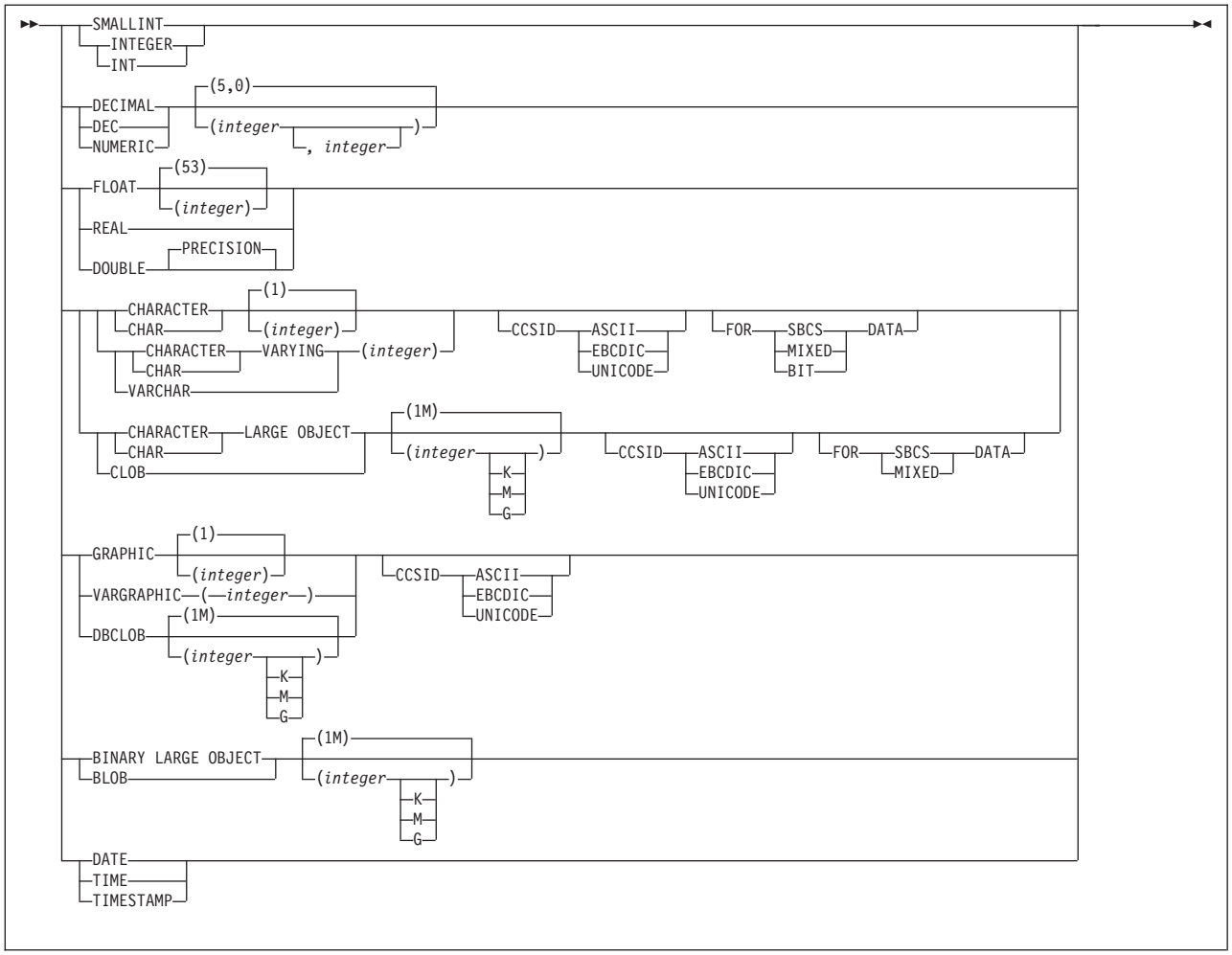
parameter-declaration:



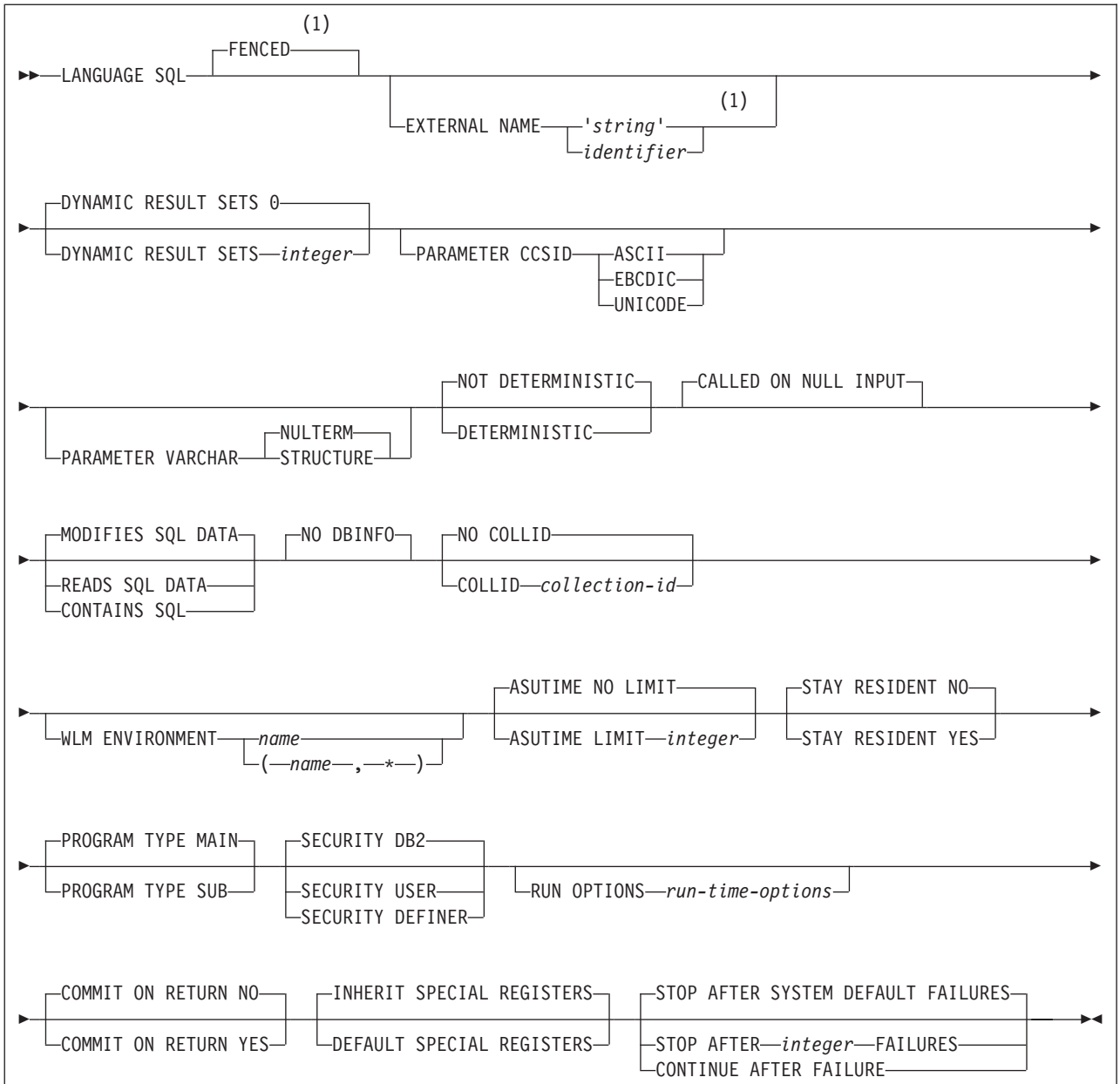
parameter-type:



built-in-type:



option-list: (The options can be specified in any order, but each option can be specified only one time)



Notes:

1 Either the FENCED or EXTERNAL NAME clause must be specified to indicate that the definition is for an external SQL procedure.

Description

procedure-name

Names the procedure. The name, including the implicit or explicit qualifier, must not identify an existing stored procedure at the current server.

(*parameter-declaration***,...)**

Specifies the number of parameters of the procedure, the data type of each parameter, and the name of each parameter. A parameter for a procedure can be used only for input, only for output, or for both input and output. If an

error is returned by the procedure, OUT parameters are undefined, and INOUT parameters are unchanged. All of the parameters are nullable.

IN Identifies the parameter as an input parameter to the procedure. The value of the parameter on entry to the procedure is the value that is returned to the calling SQL application, even if changes are made to the parameter within the procedure.

IN is the default.

OUT
Identifies the parameter as an output parameter that is returned by the procedure. If the parameter is not set within the procedure, the null value is returned.

INOUT
Identifies the parameter as both an input and output parameter for the procedure. If the parameter is not set within the procedure, its input value is returned.

parameter-name
Names the parameter for use as an SQL variable. *parameter-name* is an SQL identifier and must not be a delimited identifier that includes lowercase letters or special characters. A parameter name cannot be the same as the name of any other parameter for this version of the procedure.

parameter-type
Specifies the data type of the parameter.

built-in-type
The data type of the parameter is a built-in data type.

For more information on the data types, including the subtype of character data types (the FOR *subtype* DATA clause), see built-in-type. For external SQL procedures, the maximum limit for VARCHAR is 32767 and for VARGRAPHIC is 16382.

For parameters with a character or graphic data type, the PARAMETER CCSID clause or CCSID clause indicates the encoding scheme of the parameter. If you do not specify either of these clauses, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

Although an input parameter with a character data type has an implicitly or explicitly specified subtype (BIT, SBCS, or MIXED), the value that is actually passed in the input parameter can have any subtype. Therefore, conversion of the input data to the subtype of the parameter might occur when the procedure is called. With ASCII or EBCDIC, an error occurs if mixed data that actually contains DBCS characters is used as the value for an input parameter that is declared with an SBCS subtype.

A parameter with a datetime data type is passed to the SQL procedure as a character data type, and the data is passed in ISO format.

The encoding scheme for a datetime type parameter is determined as follows:

- If there are one or more parameters with a character or graphic data type, the encoding scheme of the datetime type parameter is the same as the encoding scheme of the character or graphic parameters.

- Otherwise, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

TABLE LIKE *table-name* **AS LOCATOR**
Specifies that the parameter is a transition table. However, when the procedure is called, the actual values in the transition table are not passed to the procedure. A single value is passed instead. This single value is a locator to the table, which the procedure uses to access the columns of the transition table. A procedure with a table parameter can only be invoked from the triggered action of a trigger.

The transition table includes columns that are defined as implicitly hidden in the table. The table that is identified can contain XML columns; however, the procedure cannot reference those XML columns.

For more information about the TABLE LIKE clause, see TABLE LIKE. For more information about using table locators, see *DB2 Application Programming and SQL Guide*.

LANGUAGE
Specifies the application programming language in which the procedure is written.

SQL
The procedure is written in DB2 SQL procedural language.

FENCED
Specifies that the procedure runs in an external address space. FENCED also specifies that the SQL procedure program is an MVS load module with an external name.

DYNAMIC RESULT SETS *integer*
Specifies the maximum number of query result sets that the procedure can return. The default is DYNAMIC RESULT SETS 0, which indicates that the procedure can return no result sets. The value of *integer* must be between 0 and 32767.

PARAMETER CCSID
Indicates whether the encoding scheme for character and graphic string parameters is ASCII, EBCDIC, or UNICODE. The default encoding scheme is the value that is specified in the CCSID clauses of the parameter list or in the field DEF ENCODING SCHEME on installation panel DSNTIPF.

This clause provides a convenient way to specify the encoding scheme for character and graphic string parameters. If individual CCSID clauses are specified for individual parameters in addition to this PARAMETER CCSID clause, the value that is specified in *all* of the CCSID clauses must be the same value that is specified in this clause.

This clause also specifies the encoding scheme that is to be used for system-generated parameters of the routine such as message tokens and DBINFO.

PARAMETER VARCHAR
Specifies that the representation of the values of varying length character string-parameters for procedures that specify LANGUAGE C.

NULTERM
Specifies that variable length character string parameters are represented in a NUL-terminated string form.

NULTERM is the default.

STRUCTURE
Specifies that variable length character string parameters are represented in a VARCHAR structure form.

The PARAMETER VARCHAR clause only applies to parameters in the parameter list of a procedure and in the RETURNS clause. It does not apply to system-generated parameters of the routine such as message tokens and DBINFO.

In a data sharing environment, you should not specify the PARAMETER VARCHAR clause until all members of the data sharing group support the clause. If some group members support this clause and others do not, and PARAMETER VARCHAR is specified, the routine will encounter different parameter forms depending on which group member invokes the routine.

EXTERNAL NAME *'string'* **or** *identifier*
Specifies the name of the MVS load module for the program that runs when the procedure name is specified in an SQL CALL statement. The value must conform to the naming conventions for MVS load modules: the value must be less than or equal to 8 bytes, and it must conform to the rules for an ordinary identifier with the exception that it must not contain an underscore.

EXTERNAL NAME *procedure-name* is the default. In some cases, the default name will not be valid. To avoid an invalid name, specify EXTERNAL NAME for a procedure that has a name that is greater than 8 bytes in length, contains an underscore, or does not conform to the rules for an ordinary identifier.

NOT DETERMINISTIC or DETERMINISTIC
Specifies whether the procedure returns the same results each time the procedure is called with the same IN and INOUT arguments.

NOT DETERMINISTIC
The procedure might not return the same result each time the procedure is called with the same IN and INOUT arguments, even when the referenced data in the database has not changed.

NOT DETERMINISTIC is the default.

DETERMINISTIC
The procedure always returns the same results each time the stored procedure is called with the same IN and INOUT arguments, if the referenced data in the database has not changed.

DB2 does not verify that the procedure code is consistent with the specification of DETERMINISTIC or NOT DETERMINISTIC.

CALLED ON NULL INPUT
Specifies that the procedure is to be called even if any or all argument values are null, which means that the procedure must be coded to test for null argument values. The procedure can return null or non-null values.

CALLED ON NULL INPUT is the default.

MODIFIES SQL DATA, READS SQL DATA, or CONTAINS SQL
Specifies the classification of SQL statements that the procedure can execute. For the data access classification of each statement, see Table 162 on page 2030. Statements that are not supported in any procedure return an error.

MODIFIES SQL DATA
Specifies that the procedure can execute any SQL statement except statements that are not supported in procedures.

MODIFIES SQL DATA is the default.

READS SQL DATA
Specifies that the procedure can execute statements with a data access indication of READS SQL DATA or CONTAINS SQL. The procedure cannot execute SQL statements that modify data.

CONTAINS SQL
Specifies that the procedure can execute only SQL statements with a data access indication of CONTAINS SQL. The procedure cannot execute statements that read or modify data.

NO DBINFO
Specifies that no additional status information that is known by DB2 is passed to the procedure when it is invoked.

NO COLLID or COLLID *collection-id*
Identifies the package collection that is to be used when the procedure is executed. This is the package collection into which the DBRM that is associated with the procedure is bound.

NO COLLID
Specifies that the package collection for the procedure is the same as the package collection of the calling program. If the invoking program does not use a package, DB2 resolves the package by using the CURRENT PACKAGE PATH special register, the CURRENT PACKAGESET special register, or the PKLIST bind option (in this order). For details about how DB2 uses these three items, see the information on package resolution in *DB2 Application Programming and SQL Guide*.

NO COLLID is the default.

COLLID *collection-id*
Specifies the package collection for the procedure.

WLM ENVIRONMENT *name* **or (***name***,*)**
Identifies the WLM (workload manager) environment in which the stored procedure is to run when the DB2 stored procedure address space is WLM-established. The *name* of the WLM environment is an SQL identifier.

If you do not specify WLM ENVIRONMENT, the procedure runs in the default WLM-established stored procedure address space that is specified at installation time.

name
The WLM environment in which the procedure must run. If another procedure or a user-defined function calls the procedure and that calling routine is running in an address space that is not associated with the specified WLM environment, DB2 routes the procedure request to a different address space.

(name,*)
When an SQL application program directly calls a procedure, *name* specifies the WLM environment in which the procedure runs.

If another procedure or a user-defined function calls the stored procedure, the procedure runs in the same WLM environment that the calling routine uses.

To define a procedure that is to run in a specified WLM environment, you must have appropriate authority for the WLM environment. For an example of a RACF command that provides this authorization, see Running stored procedures.

ASUTIME
Specifies the total amount of processor time, in CPU service units, that a single invocation of a procedure can run. The value is unrelated to the ASUTIME column of the resource limit specification table.

When you are debugging a procedure, setting a limit can be helpful in case the procedure gets caught in a loop. For information on service units, see *z/OS MVS Initialization and Tuning Guide*.

NO LIMIT
There is no limit on the number of CPU service units that the procedure can run.

NO LIMIT is the default.

LIMIT *integer*
The limit on the number of CPU service units is a positive *integer* in the range of 1 to 2 147 483 647. If the procedure uses more service units than the specified value, DB2 cancels the procedure. The CPU cycles that are consumed by parallel tasks in a procedure do not contribute towards the specified ASUTIME LIMIT.

STAY RESIDENT
Specifies whether the load module for the procedure remains resident in memory when the procedure ends.

NO The load module is deleted from memory after the procedure ends.

NO is the default.

YES
The load module remains resident in memory after the procedure ends.

PROGRAM TYPE
Specifies whether the procedure runs as a main routine or a subroutine.

MAIN
The procedure runs as a main routine.

MAIN is the default.

SUB
The procedure runs as a subroutine.

SECURITY
Specifies how the procedure interacts with an external security product, such as RACF, to control access to non-SQL resources.

DB2
The procedure does not require a special external security environment. If the procedure accesses resources that an external security product protects, the access is performed using the authorization ID that is associated with the address space in which the procedure runs.

DB2 is the default.

USER
An external security environment should be established for the procedure.

If the procedure accesses resources that the external security product protects, the access is performed using the authorization ID of the user who invoked the procedure.

DEFINER
An external security environment should be established for the procedure. If the procedure accesses resources that the external security product protects, the access is performed using the authorization ID of the owner of the procedure.

RUN OPTIONS *run-time-options*
Specifies the Language Environment run time options that are to be used for the procedure. You must specify *run-time-options* as a character string that is no longer than 254 bytes. If you do not specify RUN OPTIONS or pass an empty string, DB2 does not pass any run time options to Language Environment, and Language Environment uses its installation defaults.

For a description of the Language Environment run time options, see *z/OS Language Environment Programming Reference*.

COMMIT ON RETURN
Indicates whether DB2 commits the transaction immediately on return from the procedure.

NO DB2 does not issue a commit when the procedure returns.

NO is the default.

YES
DB2 issues a commit when the procedure returns if the following statements are true:

- A positive SQLCODE is returned by the CALL statement.
- The procedure is not in a must abort state.

The commit operation includes the work that is performed by the calling application process and the procedure.

If the procedure returns result sets, the cursors that are associated with the result sets must have been defined as WITH HOLD to be usable after the commit.

INHERIT SPECIAL REGISTERS or DEFAULT SPECIAL REGISTERS
Specifies how special registers are set on entry to the routine.

INHERIT SPECIAL REGISTERS
Specifies that the values of special registers are inherited, according to the rules that are listed in the table for characteristics of special registers in a procedure in Table 40 on page 205.

INHERIT SPECIAL REGISTERS is the default.

DEFAULT SPECIAL REGISTERS
Specifies that special registers are initialized to the default values, as indicated by the rules in the table for characteristics of special registers in a procedure in Table 40 on page 205.

STOP AFTER SYSTEM DEFAULT FAILURES, STOP AFTER *nn* **FAILURES, or CONTINUE AFTER FAILURE**
Specifies the routine is stopped after failures.

STOP AFTER SYSTEM DEFAULT FAILURES
Specifies that this routine should be placed in a stopped state after the number of failures indicated by the value of field MAX ABEND COUNT on installation panel DSNTIPX.

STOP AFTER SYSTEM DEFAULT FAILURES is the default.

STOP AFTER *nn* **FAILURES**
Specifies that this routine should be placed in a stopped state after *nn* failures. The value *nn* can be an integer from 1 to 32767.

CONTINUE AFTER FAILURE
Specifies that this routine should not be placed in a stopped state after any failure.

SQL-routine-body
Specifies the statements that define the body of the SQL procedure. For information on the SQL control statements that are supported in external SQL procedures, see "SQL control statements for external SQL procedures" on page 2032.

Notes

See "Notes" on page 1335 for information about:

- Owner privileges
- Choosing data types for parameters
- Specifying the encoding scheme for parameters
- Environments for running stored procedures
- Accessing result sets from nested stored procedures

Error handling in SQL procedures: You should consider the possible exceptions that can occur for each SQL statement in the body of a procedure. Any exception SQLSTATE that is not handled within the procedure using a handler within a compound statement results in the exception SQLSTATE being returned to the caller of the procedure.

Stored procedures and user-defined session global variables:
The content of user-defined session global variables that are referenced in routines is inherited from the caller. User-defined session global variables can be modified in stored procedures, except when the stored procedure is called by a trigger or a function.

If the procedure contains references to user-defined session global variables, the level of SQL data access must be at least CONTAINS SQL. If the procedure contains SQL statements that modify user-defined session global variables, the level of SQL data access must be MODIFIES SQL DATA.

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords:

- RESULT SET and RESULT SETS as synonyms for DYNAMIC RESULT SETS
- VARIANT as a synonym for NOT DETERMINISTIC
- NOT VARIANT as a synonym for DETERMINISTIC

Examples

Example 1: Create the definition for an SQL procedure. The procedure accepts an employee number and a multiplier for a pay raise as input. The following tasks are performed in the procedure body:

- Calculate the employee's new salary.
- Update the employee table with the new salary value.

```
CREATE PROCEDURE UPDATESALARY
 (IN EMPLOYEE_NUMBER CHAR(10),
 IN RATE DECIMAL(6,2))
 LANGUAGE SQL
 FENCED
 EXTERNAL NAME 'USALARY1'
 MODIFIES SQL DATA
  UPDATE EMP
  SET SALARY = SALARY * RATE
  WHERE EMPNO = EMPLOYEE_NUMBER
```

Example 2: Create the definition for the SQL procedure described in example 1, but specify that the procedure has these characteristics:

- The procedure runs in a WLM environment called PARTSA.
- The same input always produces the same output.
- SQL work is committed on return to the caller.
- The Language Environment run time options to be used when the SQL procedure executes are 'MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)'.

```
CREATE PROCEDURE UPDATESALARY
 (IN EMPLOYEE_NUMBER CHAR(10),
 IN RATE DECIMAL(6,2))
 LANGUAGE SQL
 FENCED
 EXTERNAL NAME 'USALARY2'
 MODIFIES SQL DATA
 WLM ENVIRONMENT PARTSA
 DETERMINISTIC
 RUN OPTIONS 'MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)'
 COMMIT ON RETURN YES
   UPDATE EMP
   SET SALARY = SALARY * RATE
   WHERE EMPNO = EMPLOYEE_NUMBER
```

For more examples of SQL procedures, see “SQL control statements for external SQL procedures” on page 2032.

CREATE PROCEDURE (SQL - native)

The CREATE PROCEDURE statement defines an SQL procedure at the current server and specifies the source statements for the procedure. You can define multiple versions of the procedure. CREATE PROCEDURE is used to define the initial version, and ALTER PROCEDURE is used to define subsequent versions.

For information about the SQL control statements that are supported in native SQL procedures, refer to Chapter 6, “SQL control statements for SQL routines,” on page 1963.

Invocation

This statement can only be dynamically prepared, and the DYNAMICRULES run behavior must be specified implicitly or explicitly.

Authorization

The privilege set that is defined below must include at least one of the following:

- The CREATEIN privilege on the schema and the required authorization to add a new package or a new version of an existing package depending on the value of the BIND NEW PACKAGE field on installation panel DSNTIPP
- SYSADM authority
- SYSCTRL authority
- System DBADM

The authorization ID that matches the schema name implicitly has the CREATEIN privilege on the schema.

If a user-defined type is referenced (as the data type of a parameter or SQL variable), the privilege set must also include at least one of the following privileges or authorities:

- Ownership of the user-defined type
- The USAGE privilege on the user-defined type
- SYSADM authority

If the procedure uses a table as a parameter, the privilege set must also include at least one of the following privileges or authorities:

- Ownership of the table
- The SELECT privilege on the table
- SYSADM authority

If the authorization ID that is used to create the procedure has installation SYSADM authority, the procedure is identified as system-defined procedure.

Privilege set: The privilege set is the privileges that are held by the SQL authorization ID of the process unless the process is within a trusted context and the ROLE AS OBJECT OWNER clause is specified. In that case, the privileges set is the privileges that are held by the role that is associated with the primary authorization ID of the process and the owner is that role.

If the statement is not running in a trusted context for which the ROLE AS OBJECT OWNER clause is specified, the privilege set is the set of privileges that

are held by the SQL authorization ID of the process. If the schema name is not the same as the SQL authorization ID of the process, one of the following conditions must be met:

- The privilege set includes SYSADM or SYSCTRL authority.
- The SQL authorization ID of the process has the CREATEIN privilege on the schema.

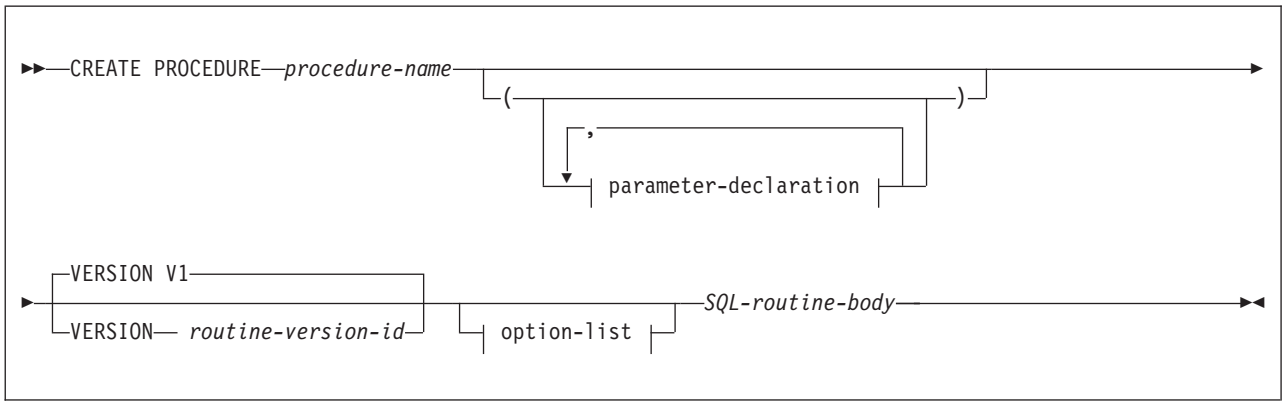
The privilege set must also include the privileges required to execute the statements in *SQL-routine-body*.

If the WLM ENVIRONMENT FOR DEBUG MODE clause is specified, the privilege set must have authority to define programs that run in the specified WLM environment.

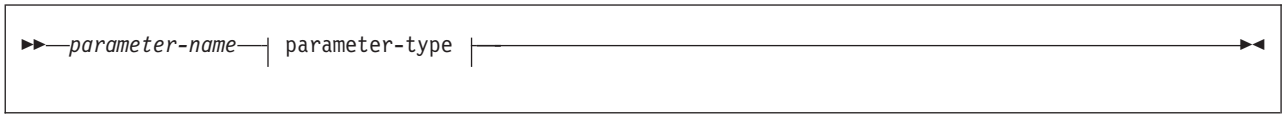
The owner of the procedure is the SQL authorization ID in the CURRENT SQLID special register unless the process is running within a trusted context and the ROLE AS OBJECT OWNER clause is specified. In that case, the owner of the procedure is the role that is associated with the primary authorization ID of the process.

The owner is implicitly given the EXECUTE privilege with the GRANT option for the procedure.

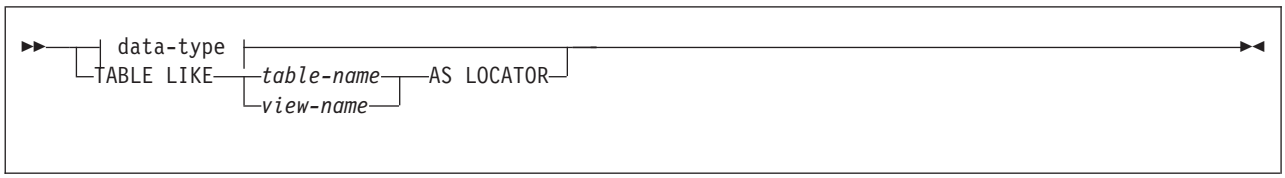
Syntax



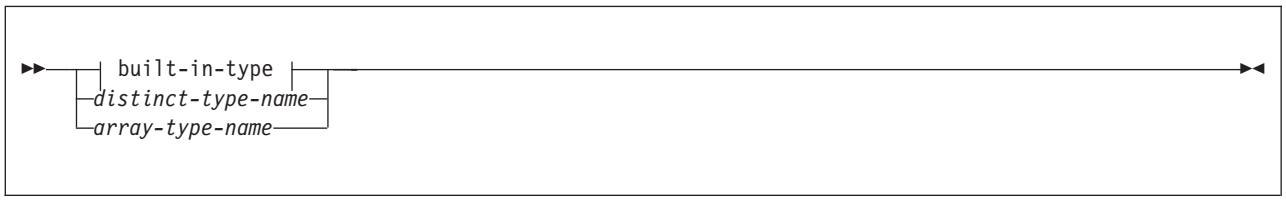
parameter-declaration:



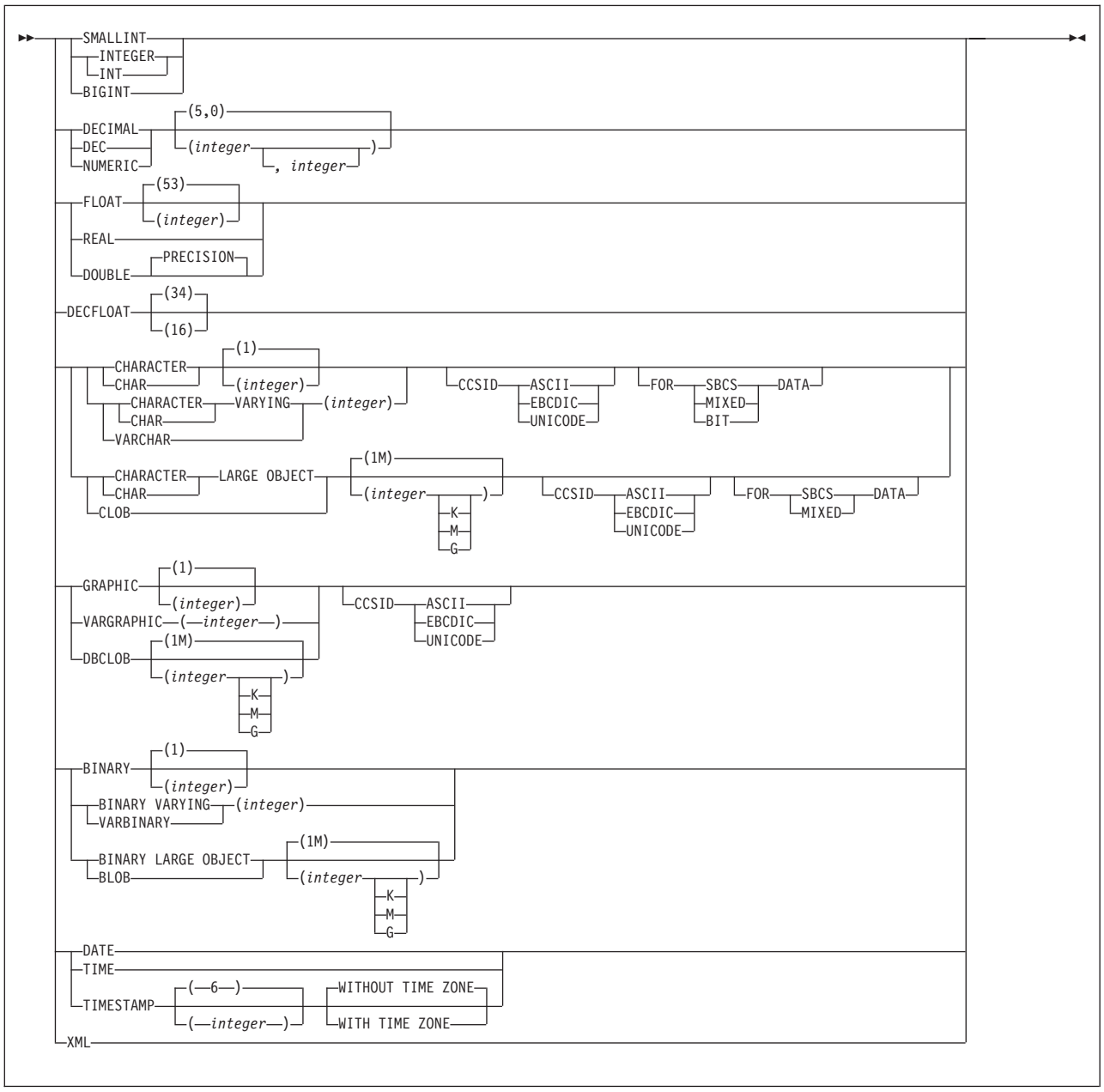
parameter-type:



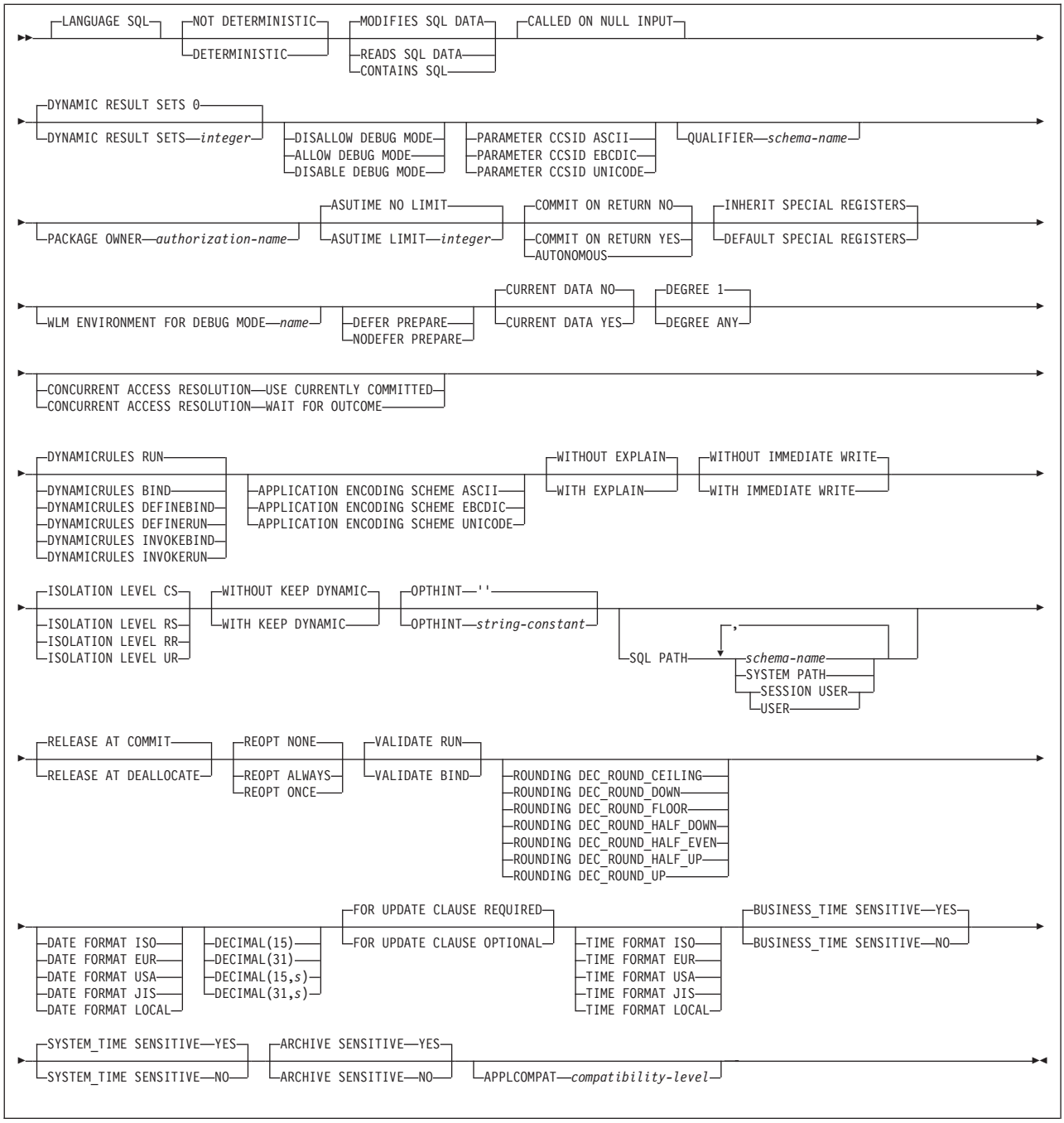
data-type:



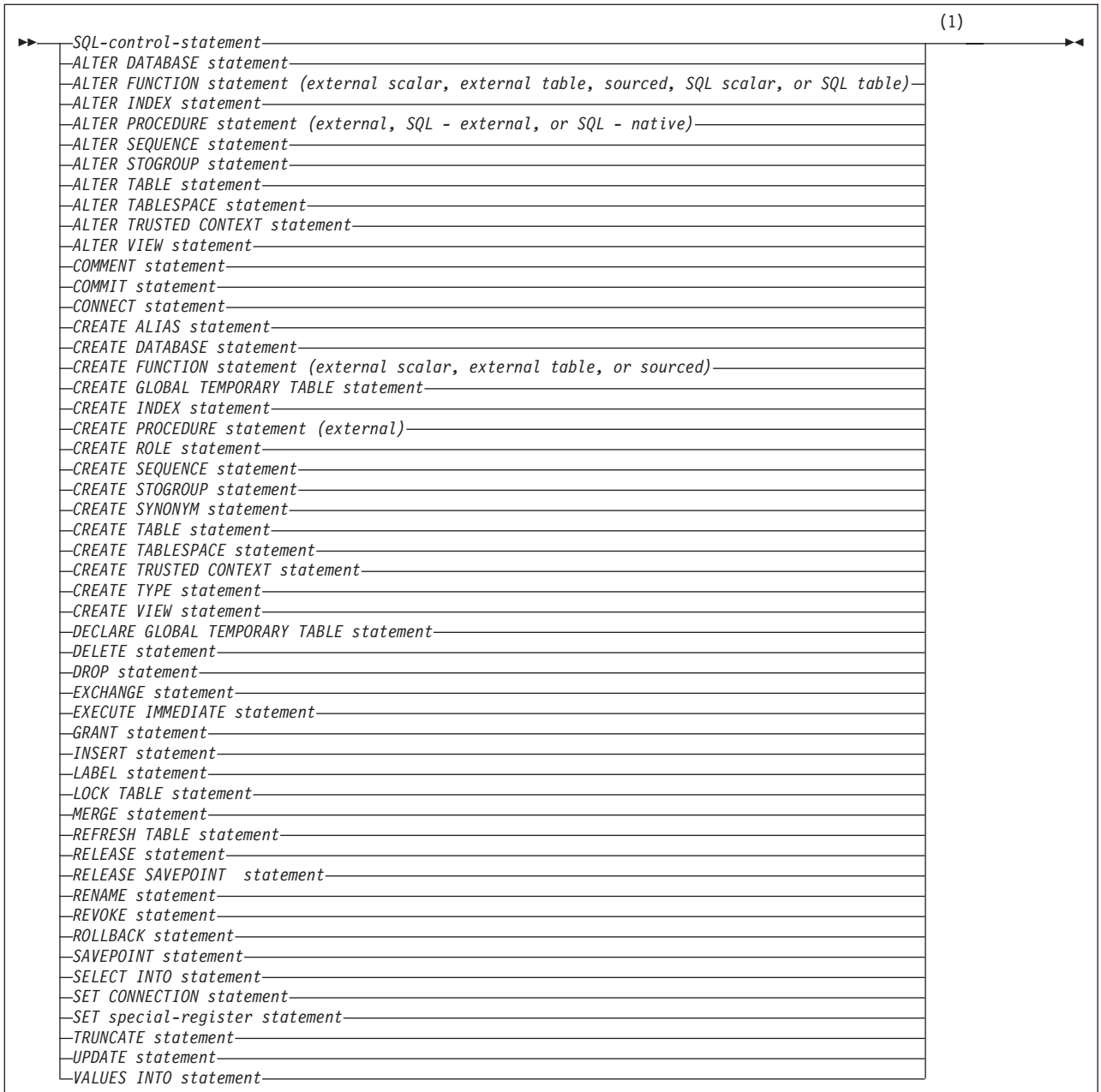
built-in-type:



option-list: (The options can be specified in any order, but each one can only be specified one time.)



SQL-routine-body:



Notes:

1 An ALTER FUNCTION statement (SQL scalar) or an ALTER PROCEDURE statement (SQL native) with an ADD VERSION or REPLACE clause are not allowed in an *SQL-routine-body*.

Description

procedure-name
Names the procedure. If *procedure-name* already exists, an error is returned even if VERSION is specified with a *version-id* that is different from any existing version identifier for the procedure that is specified in *procedure-name*.

(*parameter-declaration***,...)**
Specifies the number of parameters of the procedure, the data type and usage of each parameter, and the name of each parameter for the version of the

procedure that is being defined. The number of parameters and the specified data type and usage of each parameter must match the data types in the corresponding position of the parameter for all other versions of this procedure. Synonyms for data types are considered to be a match.

IN, OUT, and INOUT specify the usage of the parameter. The usage of the parameters must match the implicit or explicit usage of the parameters of other versions of the same procedure.

IN Identifies the parameter as an input parameter to the procedure. The value of the parameter on entry to the procedure is the value that is returned to the calling SQL application, even if changes are made to the parameter within the procedure.

IN is the default.

OUT
Identifies the parameter as an output parameter that is returned by the procedure. If the parameter is not set within the procedure, the null value is returned.

INOUT
Identifies the parameter as both an input and output parameter for the procedure. If the parameter is not set within the procedure, its input value is returned.

parameter-name
Names the parameter for use as an SQL variable. A parameter name cannot be the same as the name of any other parameter for this version of the procedure. The name of the parameter in this version of the procedure can be different than the name of the corresponding parameter for other versions of this procedure.

built-in-type
The data type of the parameter is a built-in data type.

For more information on the data types, including the subtype of character data types (the FOR *subtype* DATA clause), see built-in-type. However, the varying length string data types have different maximum lengths than for the CREATE TABLE statement. The maximum lengths for parameters (and SQL variables) are as follows: 32704 for VARCHAR or VARBINARY, and 16352 for VARGRAPHIC.

For parameters with a character or graphic data type, the PARAMETER CCSID clause or CCSID clause indicates the encoding scheme of the parameter. If you do not specify either of these clauses, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

Although an input parameter with a character data type has an implicitly or explicitly specified subtype (BIT, SBCS, or MIXED), the value that is actually passed in the input parameter can have any subtype. Therefore, conversion of the input data to the subtype of the parameter might occur when the procedure is called. With ASCII or EBCDIC, an error occurs if mixed data that actually contains DBCS characters is used as the value for an input parameter that is declared with an SBCS subtype.

Parameters with a datetime data type or a distinct type are passed to the function as a different data type:

- A datetime type parameter is passed as a character data type, and the data is passed in ISO format. The encoding scheme for a datetime type

parameter is the same as the implicitly or explicitly specified encoding scheme of any character or graphic string parameters. If no character or graphic string parameters are passed, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

- A distinct type parameter is passed as the source type of the distinct type.

AS LOCATOR
Specifies that a locator to the value of the parameter is passed to the procedure instead of the actual value. Specify AS LOCATOR only for parameters with a LOB data type or a distinct type based on a LOB data type. Passing locators instead of values can result in fewer bytes being passed to the procedure, especially when the value of the parameter is very large.

The AS LOCATOR clause has no effect on determining whether data types can be promoted.

distinct-type-name
The data type of the input parameter is a distinct type. Any length, precision, scale, subtype, or encoding scheme attributes for the parameter are those of the source type of the distinct type. The distinct type must not be based on a LOB data type.

array-type-name
The data type of the input parameter is a user-defined array type.

If you specify *array-type-name* without a schema name, DB2 resolves the array type by searching the schemas in the SQL path.

TABLE LIKE *table-name* **AS LOCATOR**
Specifies that the parameter is a transition table. However, when the procedure is called, the actual values in the transition table are not passed to the procedure. A single value is passed instead. This single value is a locator to the table, which the procedure uses to access the columns of the transition table. The table that is identified can contain XML columns; however, the procedure cannot reference those XML columns. A procedure with a table parameter can only be invoked from the triggered action of a trigger. For additional information about using table locators, refer to *DB2 Application Programming and SQL Guide*.

VERSION *routine-version-id*
Specifies the version identifier for the first version of the procedure that is to be generated. See "Naming conventions" on page 57 for information about specifying *routine-version-id*. You can use an ALTER PROCEDURE statement with the ADD VERSION clause or the BIND DEPLOY command to create additional versions of the procedure.

V1 is the default version identifier.

See Versions of a procedure for more information about the use of versions for procedures.

LANGUAGE SQL
Specifies that the procedure is written in the DB2 SQL procedural language.

DETERMINISTIC or NOT DETERMINISTIC
Specifies whether the procedure returns the same results each time it is called with the same IN and INOUT arguments.

DETERMINISTIC
The procedure always returns the same results each time it is called with the same IN and INOUT arguments if the data that is referenced in the database has not changed.

NOT DETERMINISTIC
The procedure might not return the same result each time it is called with the same IN and INOUT arguments, even when the data that is referenced in the database has not changed.

NOT DETERMINISTIC is the default.

DB2 does not verify that the procedure code is consistent with the specification of DETERMINISTIC or NOT DETERMINISTIC.

MODIFIES SQL DATA, READS SQL DATA, or CONTAINS SQL
Specifies the classification of SQL statements that the procedure can execute.

MODIFIES SQL DATA
Specifies that the procedure can execute any SQL statement except statements that are not supported in procedures.

MODIFIES SQL DATA is the default.

READS SQL DATA
Specifies that the procedure can execute statements with a data access indication of READS SQL DATA or CONTAINS SQL. The procedure cannot execute SQL statements that modify data.

CONTAINS SQL
Specifies that the procedure can execute only SQL statements with a data access indication of CONTAINS SQL. The procedure cannot execute statements that read or modify data.

CALLED ON NULL INPUT
Specifies that the procedure will be called if any, or even if all parameter values are null.

DYNAMIC RESULT SETS *integer*
Specifies the maximum number of query result sets that the procedure can return. The default is DYNAMIC RESULT SETS 0, which indicates that there are no result sets. The value must be between 0 and 32767.

ALLOW DEBUG MODE, DISALLOW DEBUG MODE, or DISABLE DEBUG MODE
Specifies whether this version of the routine can be run in debugging mode. The default is determined using the value of the CURRENT DEBUG MODE special register.

ALLOW DEBUG MODE
Specifies that this version of the routine can be run in debugging mode. When this version of the routine is invoked and debugging is attempted, a WLM environment must be available.

DISALLOW DEBUG MODE
Specifies that this version of the routine cannot be run in debugging mode.

You can use an ALTER statement to change this option to ALLOW DEBUG MODE for this initial version of the routine.

DISABLE DEBUG MODE
Specifies that this version of the routine can never be run in debugging mode.

This version of the routine cannot be changed to specify ALLOW DEBUG MODE or DISALLOW DEBUG MODE after this version of the routine has been created or altered to use DISABLE DEBUG MODE. To change this option, drop the routine and create it again using the option that you want. An alternative to dropping and recreating the routine is to create a version of the routine that uses the option that you want and making that version the active version.

When DISABLE DEBUG MODE is in effect, the WLM ENVIRONMENT FOR DEBUG MODE is ignored.

PARAMETER CCSID
Indicates whether the encoding scheme for character or graphic string parameters is ASCII, EBCDIC, or UNICODE. The default encoding scheme is the value that is specified in the CCSID clauses of the parameter list or in the field DEF ENCODING SCHEME on installation panel DSNTIPF.

This clause provides a convenient way to specify the encoding scheme for character or graphic string parameters. If individual CCSID clauses are specified for individual parameters in addition to this PARAMETER CCSID clause, the value that is specified in *all* of the CCSID clauses must be the same value that is specified in this clause.

If the data type for a parameter is a user-defined distinct type that is defined as a character or graphic type string, the CCSID of the distinct type must be the same as the value that is specified in this clause.

If the data type for a parameter is a user-defined array type that is defined with character or graphic string array elements, or a character string array index, the CCSID of these array attributes must be the same as the value that is specified in this clause.

This clause also specifies the encoding scheme that will be used for system-generated parameters of the routine.

QUALIFIER *schema-name*
Specifies the implicit qualifier that is used for unqualified names of tables, views, indexes, and aliases that are referenced in the routine body. The default value is the same as the default schema.

PACKAGE OWNER *authorization-name*
Specifies the owner of the package that is associated with the first version of the routine. The SQL authorization ID of the process is the default value.

The authorization ID must have the privileges that are required to execute the SQL statements that are contained in the routine body and must contain the necessary bind privileges. The value of PACKAGE OWNER is subject to translation when it is sent to a remote system.

If the privilege set lacks SYSADM or SYSCTRL authority, *authorization-name* must be the same as one of the authorization IDs of the process or the authorization ID of the process. If the privilege set includes SYSADM or SYSCTRL authority, *authorization-name* can be any authorization ID that contains the necessary bind privileges.

ASUTIME
Specifies the total amount of processor time, in CPU service units, that a single invocation of a routine can run. The value is unrelated to the ASUTIME column of the resource limit specification table.

When you are debugging a routine, setting a limit can be helpful in case the routine gets caught in a loop. For information on service units, see *z/OS MVS Initialization and Tuning Guide*.

NO LIMIT
Specifies that there is no limit on the service units.

NO LIMIT is the default.

LIMIT *integer*
The limit on the number of CPU service units is a positive *integer* in the range of 1 to 2 147 483 647. If the procedure uses more service units than the specified value, DB2 cancels the procedure. The CPU cycles that are consumed by parallel tasks in a procedure do not contribute towards the specified ASUTIME LIMIT.

COMMIT ON RETURN NO, COMMIT ON RETURN YES, or AUTONOMOUS
Indicates whether DB2 commits the transaction immediately on return from the procedure.

COMMIT ON RETURN NO
DB2 does not issue a commit when the procedure returns. NO is the default.

COMMIT ON RETURN YES,
DB2 issues a commit when the procedure returns if the following statements are true:

- The SQLCODE that is returned by the CALL statement is not negative.
- The procedure is not in a must-abort state.

The commit operation includes the work that is performed by the calling application process and by the procedure.

If the procedure returns result sets, the cursors that are associated with the result sets must have been defined as WITH HOLD to be usable after the commit.

AUTONOMOUS
DB2 executes the SQL procedure in a unit of work that is independent from the calling application. When this option is specified the procedure follows the rules of the COMMIT ON RETURN YES option before returning to the calling application. However, it does not commit changes in the calling application. When autonomous is specified:

- DYNAMIC RESULT SETS 0 must be in effect.
- Stored procedure parameters must not be defined as:
  - A LOB type
  - The XML data type
  - A distinct data type that is based on a LOB or XML value
  - An array type that is defined with array elements that are a LOB type

A value must not be assigned to a global variable when an autonomous procedure is executing.

INHERIT SPECIAL REGISTERS or DEFAULT SPECIAL REGISTERS
Specifies how special registers are set on entry to the routine.

INHERIT SPECIAL REGISTERS
Specifies that the values of special registers are inherited, according to the rules that are listed in the table for characteristics of special registers in a routine in Table 40 on page 205.

INHERIT SPECIAL REGISTERS is the default.

DEFAULT SPECIAL REGISTERS
Specifies that special registers are initialized to the default values, as indicated by the rules in the table for characteristics of special registers in a routine in Table 40 on page 205.

WLM ENVIRONMENT FOR DEBUG MODE *name*
Specifies the WLM (workload manager) application environment that is used by DB2 when debugging the routine. The *name* of the WLM environment is an SQL identifier.

If you do not specify WLM ENVIRONMENT FOR DEBUG MODE, DB2 uses the default WLM-established stored procedure address space specified at installation time.

To define a routine that is to run in a specified WLM application environment, you must have the appropriate authority for the WLM application environment. For an example of a RACF command that provides this authorization, see Running stored procedures.

The WLM ENVIRONMENT FOR DEBUG MODE value is ignored when DISABLE DEBUG MODE is in effect.

DEFER PREPARE or NODEFER PREPARE
Specifies whether to defer preparation of dynamic SQL statements that refer to remote objects, or to prepare them immediately.

The default depends on the value in effect for the REOPT option. If REOPT NONE is in effect, the default is inherited from the plan at run time. Otherwise, the default is DEFER PREPARE.

DEFER PREPARE
Specifies that the preparation of dynamic SQL statements that refer to remote objects will be deferred.

Refer to the DEFER(PREPARE) option in *DB2 Command Reference* for considerations with distributed processing.

NODEFER PREPARE
Specifies that the preparation of dynamic SQL statements that refer to remote objects will not be deferred.

CURRENT DATA(YES) or CURRENT DATA(NO)
Specifies whether to require data currency for read-only and ambiguous cursors when the isolation level of cursor stability is in effect. CURRENT DATA also determines whether block fetch can be used for distributed, ambiguous cursors.

YES
Specifies that data currency is required for read-only and ambiguous cursors. DB2 acquired page or row locks to ensure data currency. Block fetch is ignored for distributed, ambiguous cursors.

NO Specifies that data currency is not required for read-only and ambiguous cursors. Block fetch is allowed for distributed, ambiguous cursors. Use of **CURRENT DATA**(**NO**) is not recommended if the routine attempts to dynamically prepare and execute a DELETE WHERE CURRENT OF statement against an ambiguous cursor after that cursor is opened. You receive a negative SQLCODE if your routine attempts to use a DELETE WHERE CURRENT OF statement for any of the following cursors:

- A cursor that is using block fetch

- A cursor that is using query parallelism
- A cursor that is positioned on a row that is modified by this or another application process

NO is the default.

DEGREE
Specifies whether to attempt to run a query using parallel processing to maximize performance.

1 Specifies that parallel processing should not be used.

1 is the default.

ANY
Specifies that parallel processing can be used.

CONCURRENT ACCESS RESOLUTION
Specifies the whether processing uses only committed data or whether it will wait for commit or rollback of data that is in the process of being updated.

WAIT FOR OUTCOME
Specifies that processing will wait for the commit or rollback of data that is in the process of being updated.

USE CURRENTLY COMMITTED
Specifies that processing use the currently committed version of the data when data that is in the process of being updated is encountered. **USE CURRENTLY COMMITTED** is applicable on scans that access tables that are defined in universal table spaces with row or page level lock size.

When there is lock contention between a read transaction and an insert transaction, **USE CURRENTLY COMMITTED** is applicable to scans with isolation level CS or RS. Applicable scans include intent read scans for read-only and ambiguous queries and for updatable cursors. **USE CURRENTLY COMMITTED** is also applicable to scans initiated from WHERE predicates of UPDATE or DELETE statements and the subselect of INSERT statements.

When there is lock contention is between a read transaction and a delete transaction, **USE CURRENTLY COMMITTED** is applicable to scans with isolation level CS and when CURRENTDATA(NO) is specified.

DYNAMICRULES
Specifies the values that apply, at run time, for the following dynamic SQL attributes:

- The authorization ID that is used to check authorization
- The qualifier that is used for unqualified objects
- The source for application programming options that DB2 uses to parse and semantically verify dynamic SQL statements

DYNAMICRULES also specifies whether dynamic SQL statements can include GRANT, REVOKE, ALTER, CREATE, DROP, and RENAME statements.

In addition to the value of the DYNAMICRULES clause, the run time environment of a routine controls how dynamic SQL statements behave at run time. The combination of the DYNAMICRULES value and the run time environment determines the value for the dynamic SQL attributes. That set of attribute values is called the dynamic SQL statement behavior. The following values can be specified:

RUN
Specifies that dynamic SQL statements are to be processed using run behavior.

RUN is the default.

BIND
Specifies that dynamic SQL statements are to be processed using bind behavior.

DEFINEBIND
Specifies that dynamic SQL statements are to be processed using either define behavior or bind behavior.

DEFINERUN
Specifies that dynamic SQL statements are to be processed using either define behavior or run behavior.

INVOKEBIND
Specifies that dynamic SQL statements are to be processed using either invoke behavior or bind behavior.

INVOKERUN
Specifies that dynamic SQL statements are to be processed using either invoke behavior or run behavior.

See "Authorization IDs and dynamic SQL" on page 75 for information on the effects of these options.

APPLICATION ENCODING SCHEME
Specifies the default encoding scheme for SQL variables in static SQL statements in the routine body. The value is used for defining an SQL variable in a compound statement if the CCSID clause is not specified as part of the data type, and the PARAMETER CCSID routine option is not specified.

ASCII
Specifies that the data is encoded using the ASCII CCSIDs of the server.

EBCDIC
Specifies that the data is encoded using the EBCDIC CCSIDs of the server.

UNICODE
Specifies that the data is encoded using the Unicode CCSIDs of the server.

See the ENCODING bind option in *DB2 Command Reference* for information about how the default for this option is determined.

WITH EXPLAIN or WITHOUT EXPLAIN
Specifies whether information will be provided about how SQL statements in the routine will execute.

WITHOUT EXPLAIN
Specifies that information will not be provided about how SQL statements in the routine will execute.

You can get EXPLAIN output for a statement that is embedded in a routine that is specified using WITHOUT EXPLAIN by embedding the SQL statement EXPLAIN in the routine body. Otherwise, the value of the EXPLAIN option applies to all explainable SQL statements in the routine body, and to the fullselect portion of any DECLARE CURSOR statements.

WITHOUT EXPLAIN is the default.

WITH EXPLAIN
Specifies that information will be provided about how SQL statements in the routine will execute. Information is inserted into the table *owner*.PLAN_TABLE. *owner* is the authorization ID of the owner of the routine. Alternatively, the authorization ID of the owner of the routine can have an alias as *owner*.PLAN_TABLE that points to the base table, PLAN_TABLE. *owner* must also have the appropriate SELECT and INSERT privileges on that table. WITH EXPLAIN does not obtain information for statements that access remote objects. PLAN_TABLE must have a base table and can have multiple aliases with the same table name, PLAN_TABLE, but have different schema qualifiers. It cannot be a view or a synonym and should exist before the version is added or replaced. In all inserts to *owner*.PLAN_TABLE, the value of QUERYNO is the statement number that is assigned by DB2.

The WITH EXPLAIN option also populates two optional tables if they exist: DSN_STATEMNT_TABLE and DSN_FUNCTION_TABLE. DSN_STATEMNT_TABLE contains an estimate of the processing cost for an SQL statement. See *DB2 Application Programming and SQL Guide* for more information. DSN_FUNCTION_TABLE contains information about function resolution. See *DB2 Application Programming and SQL Guide* for more information.

For a description of the tables that are populated by the WITH EXPLAIN option, see "EXPLAIN" on page 1642.

WITH IMMEDIATE WRITE or WITHOUT IMMEDIATE WRITE
Specifies whether immediate writes are to be done for updates that are made to group buffer pool dependent page sets or partitions. This option is only applicable for data sharing environments. The IMMEDWRITE subsystem parameter has no affect of this option. *DB2 Command Reference* shows the implied hierarchy of the IMMEDWRITE bind option (which is similar to this routine option) as it affects run time.

WITHOUT IMMEDIATE WRITE
Specifies that normal write activity is performed. Updated pages that are group buffer pool dependent are written at or before phase one of commit or at the end of abort for transactions that have been rolled back.

WITHOUT IMMEDIATE WRITE is the default.

WITH IMMEDIATE WRITE
Specifies that updated pages that are group buffer pool dependent are immediately written as soon as the buffer update completes. Updated pages are written immediately even if the buffer is updated during forward progress or during the rollback of a transaction. **WITH IMMEDIATE WRITE** might impact performance.

ISOLATION LEVEL RR, RS, CS, or UR
Specifies how far to isolate the routine from the effects of other running applications. For information about isolation levels, see *DB2 Performance Monitoring and Tuning Guide*.

RR Specifies repeatable read.

RS Specifies read stability.

CS Specifies cursor stability. **CS** is the default.

UR Specifies uncommitted read.

WITH KEEP DYNAMIC or WITHOUT KEEP DYNAMIC
Specifies whether DB2 keeps dynamic SQL statements after commit points.

WITHOUT KEEP DYNAMIC
Specifies that DB2 does not keep dynamic SQL statements after commit points.

WITHOUT KEEP DYNAMIC is the default.

WITH KEEP DYNAMIC
Specifies that DB2 keeps dynamic SQL statements after commit points. If you specify WITH KEEP DYNAMIC, the application does not need to prepare an SQL statement after every commit point. DB2 keeps the dynamic SQL statement until one of the following occurs:

- The application process ends
- A rollback operations occurs
- The application executes an explicit PREPARE statement with the same statement identifier as the dynamic SQL statement

If you specify WITH KEEP DYNAMIC, and the prepared statement cache is active, the DB2 subsystem keeps a copy of the prepared statement in the cache. If the prepared statement cache is not active, the subsystem keeps only the SQL statement string past a commit point. If the application executes an OPEN, EXECUTE, or DESCRIBE operation for that statement, the statement is implicitly prepared.

If you specify WITH KEEP DYNAMIC, DDF server threads that are used to execute procedures or packages that have this option in effect will remain active. Active DDF server threads are subject to idle thread timeouts, as described in *DB2 Installation Guide* for installation panel DSNTIPR.

If you specify WITH KEEP DYNAMIC, you must not specify REOPT ALWAYS. WITH KEEP DYNAMIC and REOPT ALWAYS are mutually exclusive. However, you can specify WITH KEEP DYNAMIC and REOPT ONCE.

Use WITH KEEP DYNAMIC to improve performance if your DRDA client application uses a cursor that is defined as WITH HOLD. The DB2 subsystem automatically closes a held cursor when there are no more rows to retrieve, which eliminates an extra network message.

OPTHINT *string-constant*
Specifies whether query optimization hints are used for static SQL statements that are contained within the body of the routine.

string-constant is a character string of up to 128 bytes in length, which is used by the DB2 subsystem when searching the PLAN_TABLE for rows to use as input. The default value is an empty string, which indicates that the DB2 subsystem does not use optimization hints for static SQL statements.

Optimization hints are only used if optimization hints are enabled for your system. See *DB2 Installation Guide* for information about enabling optimization hints.

SQL PATH
Specifies the SQL path that DB2 uses to resolve unqualified user-defined type, function, and procedure names in the procedure body. The default value is "SYSIBM", "SYSFUN", "SYSPROC", "SYSIBMADM", and the value of the QUALIFIER option, which is the qualifier for the procedure that is the target of the statement.

Schemas "SYSIBM", "SYSFUN", "SYSPROC", "SYSIBMADM" do not need to be explicitly specified. If any of these schemas is not explicitly specified, it is implicitly assumed at the beginning the SQL path.

DB2 calculates the length by taking each *schema-name* specified and removing any trailing blanks from it, adding two delimiters around it, and adding one comma after each schema name except for the last one. The length of the resulting string cannot exceed the length of the CURRENT SCHEMA special register. If you do not specify the "SYSIBM", "SYSFUN", "SYSPROC", "SYSIBMADM", schemas, they are not included in the length of the SQL path. If the total length of the SQL path exceeds the length of the CURRENT PATH special register, DB2 returns an error for the CREATE statement.

Related information:

"SQL path" on page 64

"CURRENT SCHEMA" on page 191

"CURRENT PATH" on page 184

schema-name
Specifies a schema. DB2 does not validate that the specified schema actually exists when the CREATE statement is processed.

SYSPUBLIC must not be specified for the SQL path.

schema-name-list
Specifies a comma separated list of schema names. The same schema name should not appear more than one time in the list of schema names. The number of schema names that you can specify is limited by the maximum length of the resulting SQL path.

SYSPUBLIC must not be specified for the SQL path.

SYSTEM PATH
Specifies the schema names "SYSIBM", "SYSFUN", "SYSPROC", "SYSIBMADM".

SESSION_USER or USER
Specifies the value of the SESSION_USER or USER special register, which represents a maximum 8 byte (in EBCDIC) *schema-name*. At the time the CREATE statement is processed, this length is included in the total length of the list of schema names that is specified for the PATH bind option.

RELEASE AT
Specifies when to release resources that the procedure uses: either at each commit point or when the procedure terminates.

COMMIT
Specifies that resources will be released at each commit point.

COMMIT is the default.

DEALLOCATE
Specifies that resources will be released only when the procedure terminates. DEALLOCATE has no effect on dynamic SQL statements, which always use RELEASE AT COMMIT, with this exception: When you use the RELEASE AT DEALLOCATE clause and the WITH KEEP DYNAMIC clause, and the subsystem is installed with a value of YES for the field CACHE DYNAMIC SQL on installation panel DSNTIP8, the RELEASE AT DEALLOCATE option is honored for dynamic SELECT and data change statements.

Locks that are acquired for dynamic statements are held unit one of the following events occurs:

- The application process ends.
- The application process issues a PREPARE statement with the same statement identifier. (Locks are released at the next commit point).
- The statement is removed from the prepared statement cache because the statement has not been used. (Locks are released at the next commit point).
- An object that the statement is dependent on is dropped or altered, or a privilege that the statement needs is revoked. (Locks are released at the next commit point).

RELEASE AT DEALLOCATE can increase the package or plan size because additional items become resident in the package or plan. For more information about how the RELEASE clause affects locking and concurrency, see *DB2 Performance Monitoring and Tuning Guide*.

REOPT

Specifies if DB2 will determine the access path at run time by using the values of SQL variables or SQL parameters, parameter markers, and special registers.

NONE

Specifies that DB2 does not determine the access path at run time by using the values of SQL variables or SQL parameters, parameter markers, and special registers.

NONE is the default.

ALWAYS

Specifies that DB2 always determines the access path at run time each time an SQL statement is run. Do not specify REOPT ALWAYS with the WITH KEEP DYNAMIC or NODEFER PREPARE clauses.

ONCE

Specifies that DB2 determine the access path for any dynamic SQL statements only once, at the first time the statement is opened. This access path is used until the prepared statement is invalidated or removed from the dynamic statement cache and need to be prepared again.

VALIDATE RUN or VALIDATE BIND

Specifies whether to recheck, at run time, errors of the type "OBJECT NOT FOUND" and "NOT AUTHORIZED" that are found during bind or rebind. The option has no effect if all objects and needed privileges exist.

VALIDATE RUN

Specifies that if needed objects or privileges do not exist when the CREATE statement is processed, warning messages are returned, but the CREATE statement succeeds. The DB2 subsystem rechecks for the objects and privileges at run time for those SQL statements that failed the checks during processing of the CREATE statement. The authorization checks the use of the authorization ID of the owner of the routine.

VALIDATE RUN is the default.

VALIDATE BIND

Specifies that if needed objects or privileges do not exist at the time the CREATE statement is processed, an error is issued and the CREATE statement fails.

ROUNDING
Specifies the rounding mode for manipulation of DECFLOAT data. The default value is taken from the DEFAULT DECIMAL FLOATING POINT ROUNDING MODE in DECP.

DEC_ROUND_CEILING
Specifies numbers are rounded towards positive infinity.

DEC_ROUND_DOWN
Specifies numbers are rounded towards 0 (truncation).

DEC_ROUND_FLOOR
Specifies numbers are rounded towards negative infinity.

DEC_ROUND_HALF_DOWN
Specifies numbers are rounded to nearest; if equidistant, round down.

DEC_ROUND_HALF_EVEN
Specifies numbers are rounded to nearest; if equidistant, round so that the final digit is even.

DEC_ROUND_HALF_UP
Specifies numbers are rounded to nearest; if equidistant, round up.

DEC_ROUND_UP
Specifies numbers are rounded away from 0.

DATE FORMAT ISO, EUR, USA, JIS, or LOCAL
Specifies the date format for result values that are string representations of date or time values. See "String representations of datetime values" on page 101 for more information.

The default format is specified in the DATE FORMAT field of installation panel DSNTIP4 of the system where the routine is defined. You cannot use the LOCAL option unless you have a date exit routine.

DECIMAL(15), DECIMAL(31), DECIMAL(15,*s*), or DECIMAL(31,*s*)
Specifies the maximum precision that is to be used for decimal arithmetic operations. See "Arithmetic with two decimal operands" on page 244 for more information. The default format is specified in the DECIMAL ARITHMETIC field of installation panel DSNTIPF of the system where the routine is defined. If the form *pp.s* is specified, *s* must be a number between 1 and 9. *s* represents the minimum scale that is to be used for division.

FOR UPDATE CLAUSE OPTIONAL or FOR UPDATE CLAUSE REQUIRED
Specifies whether the FOR UPDATE clause is required for a DECLARE CURSOR statement if the cursor is to be used to perform positioned updates.

FOR UPDATE CLAUSE REQUIRED
Specifies that a FOR UPDATE clause must be specified as part of the cursor definition if the cursor will be used to make positioned updates.

FOR UPDATE CLAUSE REQUIRED is the default.

FOR UPDATE CLAUSE OPTIONAL
Specifies that the FOR UPDATE clause does not need to be specified in order for a cursor to be used for positioned updates. The routine body can include positioned UPDATE statements that update columns that the user is authorized to update.

The FOR UPDATE clause with no column list applies to static or dynamic SQL statements. Even if you do not use this clause, you can specify FOR UPDATE

OF with a column list to restrict updates to only the columns that are named in the FOR UPDATE clause and to specify the acquisition of update locks.

TIME FORMAT ISO, EUR, USA, JIS, or LOCAL
Specifies the time format for result values that are string representations of date or time values. See "String representations of datetime values" on page 101 for more information.

The default format is specified in the TIME FORMAT field of installation panel DSNTIP4 of the system where the routine is defined. You cannot use the LOCAL option unless you have a date exit routine.

BUSINESS_TIME SENSITIVE
Determines whether references to application-period temporal tables in both static and dynamic SQL statements are affected by the value of the CURRENT TEMPORAL BUSINESS_TIME special register.

YES
References to application-period temporal tables are affected by the value of the CURRENT TEMPORAL BUSINESS_TIME special register. YES is the default value.

NO References to application-period temporal tables are not affected by the value of the CURRENT TEMPORAL BUSINESS_TIME special register.

Related information:

"CURRENT TEMPORAL BUSINESS_TIME" on page 194

SYSTEM_TIME SENSITIVE
Determines whether references to system-period temporal tables in both static and dynamic SQL statements are affected by the value of the CURRENT TEMPORAL SYSTEM_TIME special register.

YES
References to system-period temporal tables are affected by the value of the CURRENT TEMPORAL SYSTEM_TIME special register. YES is the default value.

NO References to system-period temporal tables are not affected by the value of the CURRENT TEMPORAL SYSTEM_TIME special register.

Related information:

"CURRENT TEMPORAL SYSTEM_TIME" on page 196

ARCHIVE SENSITIVE
Determines whether references to archive-enabled tables in SQL statements are affected by the value of the SYSIBMADM.GET_ARCHIVE global variable.

YES
References to archive-enabled tables are affected by the value of the SYSIBMADM.GET_ARCHIVE global variable. YES is the default value.

NO References to archive-enabled tables are not affected by the value of the SYSIBMADM.GET_ARCHIVE global variable.

Related information:

"References to built-in global variables" on page 223

APPLCOMPAT *compatibility-level*
Specifies the package compatibility level behavior for static SQL, If this option is not specified then the behavior is determined, in priority order, by the

compatibility-level of the last BIND of the package or the APPLCOMPAT system parameter. The following values of *compatibility-level* can be specified:

V10R1
The static SQL statements in the package have V10R1 compatibility behavior.

V11R1
The static SQL statements in the package have V11R1 compatibility behavior.

Related information:

APPL COMPAT LEVEL field (APPLCOMPAT subsystem parameter) (DB2 Installation and Migration)

SQL-routine-body
Specifies the statements that define the body of the SQL procedure. For information on the SQL control statements that are supported in native SQL procedures, see Chapter 6, "SQL control statements for SQL routines," on page 1963. If an *SQL-procedure-statement* is the only statement in the procedure body, the statement must not end with a semicolon.

Notes

Additional Notes:
See "Notes" on page 1335 for information about:

- Owner privileges
- Choosing data types for parameters
- Environments for running stored procedures
- Accessing result sets from nested stored procedures

Versions of a procedure:
The CREATE PROCEDURE statement for an SQL procedure defines the initial version of the procedure. You can define additional versions using the ADD VERSION clause of the ALTER PROCEDURE statement. All versions of a procedure share the same procedure signature and the same specific name. However, the parameters names can differ between versions of a procedure. Only one version of the procedure can be considered to be the active version of the procedure.

Characteristics of the package that is generated for a procedure:
The package that is associated with the first version of a procedure is named as follows:

- *location* is set to the value of the CURRENT SERVER special register
- *collection-id* (schema) for the package is the same as the schema qualifier of the procedure.
- *package-id* is the same as the specific name of the procedure
- *version-id* is the same as the version identifier for the initial version of the procedure.

If you want to change the *collection-id* for the name of the package, you need to make a copy of the package.

The package is generated using the bind options that correspond to the implicitly or explicitly specified procedure options. See "Implicit rebinding and regeneration that occurs because of an ALTER PROCEDURE statement" in Chapter 6, "SQL control statements for SQL routines," on page 1963

page 1963 for more information. In addition to the corresponding bind options, the package is generated using the following bind options:

- DBPROTOCOL(DRDA)
- FLAG(1)
- SQLERROR(NOPACKAGE)
- ENABLE(*)

See Table 98 on page 969 for additional information about the correspondence of procedure options to bind options.

Considerations for SQL processor programs:
SQL processor programs, such as SPUFI, the command line processor, and DSNTEP2, might not correctly parse SQL statements in the routine body that end with semicolons. These processor programs accept multiple SQL statements as input, with each statement separated with a terminator character. Processor programs that use a semicolon as the SQL statement terminator can truncate a CREATE FUNCTION statement with embedded semicolons and pass only a portion of it to DB2. Therefore, you might need to change the SQL terminator character for these processor programs. For information on changing the terminator character for SPUFI and DSNTEP2, see *DB2 Application Programming and SQL Guide*.

Identifier resolution:
See Chapter 6, "SQL control statements for SQL routines," on page 1963 for information on how names are resolved to columns, SQL variables, or SQL parameters within an SQL routine.

If duplicate names are used for columns and SQL variables and parameters, qualify the duplicate names by using the table designator for columns, the routine name for parameters, and the label name for SQL variables.

Lines within the SQL procedure definition:
When a procedure is created, information is retained on lines in the CREATE statement. Lines are determined by the presence of the new line control character.

Error handling in SQL procedures:
You should consider the possible exceptions that can occur for each SQL statement in the body of a procedure. Any exception SQLSTATE that is not handled within the procedure using a handler within a compound statement or a compound statement that is nested within that compound statement results in the exception SQLSTATE being returned to the caller of the procedure.

Stored procedures and user-defined session global variables:
The content of user-defined session global variables that are referenced in routines is inherited from the caller. User-defined session global variables can be modified in stored procedures, except when the stored procedure is called by a trigger or a function.

If the procedure contains references to user-defined session global variables, the level of SQL data access must be at least CONTAINS SQL. If the procedure contains SQL statements that modify user-defined session global variables, the level of SQL data access must be MODIFIES SQL DATA.

Specifying the encoding scheme for parameters:
The encoding scheme of all of the parameters with a character or graphic

string data type, distinct type with a CCSID, or an array type with a CCSID (both input and output parameters) must be the same—either all ASCII, all EBCDIC, or all UNICODE. If you specify the encoding scheme on the individual parameters, instead of using PARAMETER CCSID to specify it for all parameters at once or allowing the encoding scheme to default to the system value, ensure that all encoding schemes agree.

Stored procedures with a parameter that is defined as an array type:
A procedure that is defined with a parameter that is an array type can be invoked only from within an SQL PL context.

Compatibilities:
For compatibility with previous versions of DB2, the following clauses can be specified, but they will be ignored and a warning will be issued:

- STAY RESIDENT
- PROGRAM TYPE
- RUN OPTIONS
- NO DBINFO
- COLLID or NOCOLLID
- SECURITY
- PARAMETER STYLE GENERAL WITH NULLS
- STOP AFTER SYSTEM DEFAULT FAILURES
- STOP AFTER *nn* FAILURES
- CONTINUE AFTER FAILURES
- PARAMETER VARCHAR

If the FENCED or EXTERNAL clause is specified, an external SQL procedure will be generated. See "CREATE PROCEDURE (SQL - external)" on page 1338 for more information.

If WLM ENVIRONMENT is specified without the FOR DEBUG MODE keywords, and error is returned. If WLM ENVIRONMENT is specified for a native SQL procedure, WLM ENVIRONMENT FOR DEBUG MODE must be specified.

Alternative syntax and synonyms:
To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords:

- RESULT SET and RESULT SETS as synonyms for DYNAMIC RESULT SETS
- VARIANT as a synonym for NOT DETERMINISTIC
- NOT VARIANT as a synonym for DETERMINISTIC
- NULL CALL as a synonym for CALLED ON NULL

Examples

Example 1: Create the definition for an SQL procedure. The procedure accepts an employee number and a multiplier for a pay raise as input. The following tasks are performed in the procedure body:

- Calculate the employee's new salary.
- Update the employee table with the new salary value.

```
CREATE PROCEDURE UPDATE_SALARY_1
 (IN EMPLOYEE_NUMBER CHAR(10),
 IN RATE DECIMAL(6,2))
 LANGUAGE SQL
```

```
 MODIFIES SQL DATA
  UPDATE EMP
  SET SALARY = SALARY * RATE
  WHERE EMPNO = EMPLOYEE_NUMBER
```

Example 2: Create the definition for the SQL procedure described in example 1, but specify that the procedure has these characteristics:

- The same input always produces the same output.
- SQL work is committed on return to the caller.

```
CREATE PROCEDURE UPDATE_SALARY_1
 (IN EMPLOYEE_NUMBER CHAR(10),
 IN RATE DECIMAL(6,2))
 LANGUAGE SQL
 MODIFIES SQL DATA
 DETERMINISTIC
 COMMIT ON RETURN YES
   UPDATE EMP
   SET SALARY = SALARY * RATE
   WHERE EMPNO = EMPLOYEE_NUMBER
```

Example 3: Create the definition for an SQL procedure that uses arrays as IN and OUT parameters. The procedure is named GETWEEKENDS. It accepts an array of DATE values as input, and returns an array that contains only the dates that fall on a Saturday or Sunday. For example, if the input dates are a Saturday, a Friday, and a Sunday, the procedure returns only the dates that fall on Saturday and Sunday.

Suppose that the following user-defined array type has been defined:

```
CREATE TYPE DATEARRAY AS DATE ARRAY[100];
```

After the array type is created, any references to it need to specify the fully qualified user-defined array type name. Otherwise, the schema for the type needs to be in the CURRENT PATH.

Suppose that the SQL procedure is defined like this:

```
CREATE PROCEDURE GETWEEKENDS(IN MYDATES DATEARRAY, OUT WEEKENDS DATEARRAY)
 BEGIN
  -- ARRAY INDEX VARIABLES
  DECLARE DATEINDEX, WEEKENDINDEX INT DEFAULT 1;
  -- VARIABLE TO STORE THE ARRAY LENGTH OF MYDATES,
  -- INITIALIZED USING THE CARDINALITY FUNCTION.
  DECLARE DATESCOUNT INT;
  SET DATESCOUNT = CARDINALITY(MYDATES);
  -- FOR EACH DATE IN MYDATES, IF THE DATE IS A SUNDAY OR SATURDAY,
  -- ADD IT TO THE OUTPUT ARRAY NAMED "WEEKENDS"
  WHILE DATEINDEX <= DATESCOUNT DO
   IF DAYOFWEEK(MYDATES[DATEINDEX]) IN (1, 7) THEN
    SET WEEKENDS[WEEKENDINDEX] = MYDATES[DATEINDEX];
    SET WEEKENDINDEX = WEEKENDINDEX + 1;
   END IF;
   SET DATEINDEX = DATEINDEX + 1;
  END WHILE;
 END
```

Also suppose that input array MYDATES contains the following content:

```
['2012-04-28', '2012-02-10','2012-03-18']
```

After the procedure returns, output array WEEKENDS contains the following content:

```
['2012-04-28', '2012-03-18']
```

Example 4: Create the definition for an SQL procedure that uses arrays as OUT parameters. The procedure is named GET_PHONES. It returns an array that contains phone numbers for employee 1775. If more than five phone numbers exist for the employee, an error is returned because the array is defined for only five elements.

Suppose that the following user-defined array type and table have been defined:

```
CREATE TYPE PHONELIST AS DECIMAL(10, 0) ARRAY[5];
CREATE TABLE EMP_PHONES(ID INTEGER, PHONENUMBER DECIMAL(10,0));
```

The SQL procedure is defined like this:

```
CREATE PROCEDURE GET_PHONES(OUT EPHONES PHONELIST)
 BEGIN
  SELECT ARRAY_AGG(PHONENUMBER)
   INTO EPHONES
   FROM EMP_PHONES
   WHERE ID = 1775;
 END
```

For more examples of SQL procedures, see Chapter 6, "SQL control statements for SQL routines," on page 1963.

CREATE ROLE

The CREATE ROLE statement creates a role at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

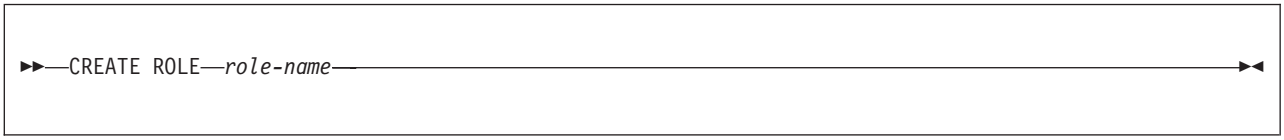
The privilege set that is defined below must include at least one of the following authorities:

- SYSADM authority
- SYSCTRL authority
- SECADM

Privilege set: If the statement is embedded in an application program, the privilege set is the set of privileges that are held by the owner of the plan or package.

If the statement is dynamically prepared, the privilege set is the set of privileges that are held by the SQL authorization ID of the process or by the role that is associated with the primary authorization ID, if the statement is run in a trusted context and the ROLE AS OBJECT OWNER clause is specified.

Syntax



Description

role-name
Names the role. The name must not identify a role that exists at the current server. The name must not begin with the characters 'SYS' and must not be 'DBADM', ‘NONE', 'NULL', 'PUBLIC', or 'SECADM'.

Examples

The following statement creates a role named TELLER.

```
CREATE ROLE TELLER;
```

CREATE SEQUENCE

The CREATE SEQUENCE statement creates a sequence at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set that is defined below must include at least one of the following:

- The CREATEIN privilege on the schema
- SYSADM or SYSCTRL authority
- System DBADM

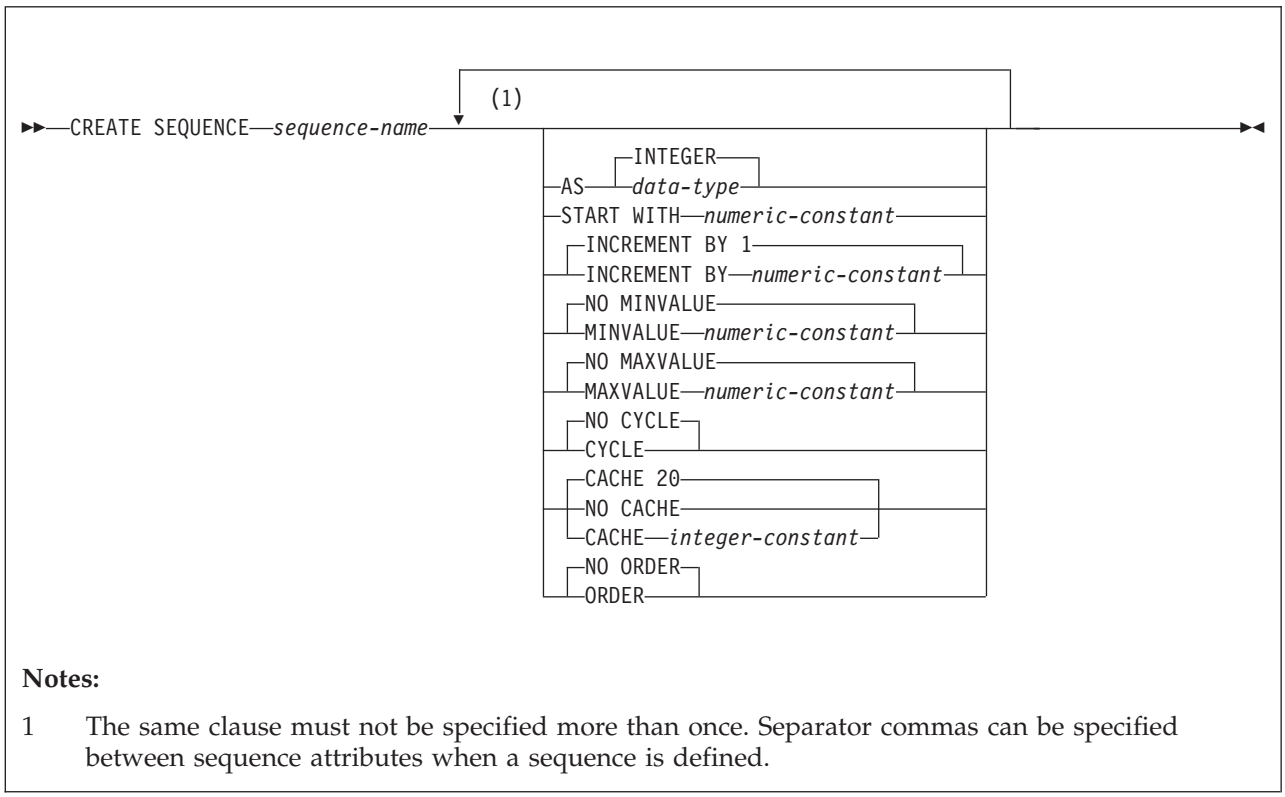
The authorization ID that matches the schema name implicitly has the CREATEIN privilege on the schema.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the of the owner of the plan or package. If the application is bound in a trusted context with the ROLE AS OBJECT OWNER clause specified, a role is the owner. Otherwise, an authorization ID is the owner.

If the statement is dynamically prepared, the privilege set is the privileges that are held by the SQL authorization ID of the process unless the process is within a trusted context and the ROLE AS OBJECT OWNER clause is specified. In that case, the privileges set is the privileges that are held by the role that is associated with the primary authorization ID of the process.

If the data type of the sequence is a distinct type, the privilege set must include the USAGE privilege on the distinct type.

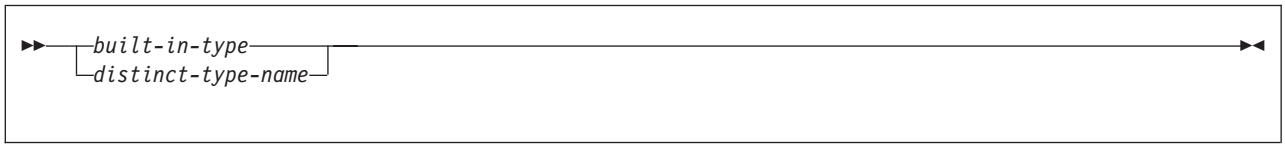
Syntax



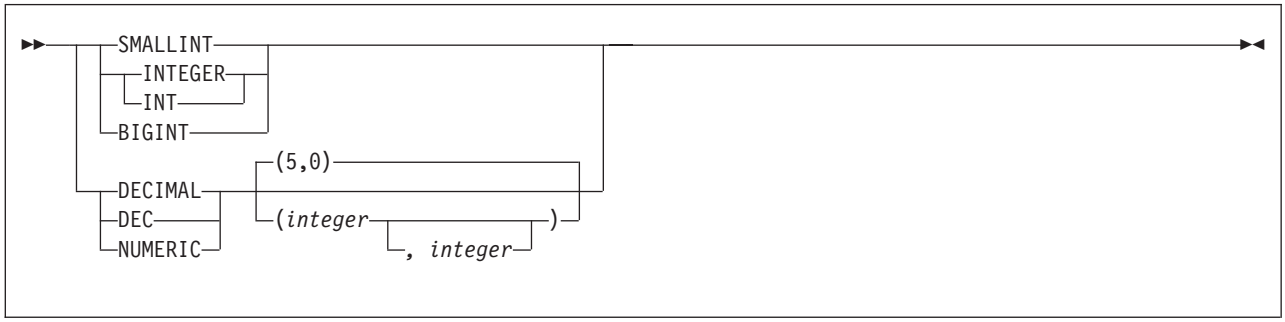
Notes:

1 The same clause must not be specified more than once. Separator commas can be specified between sequence attributes when a sequence is defined.

data-type:



built-in-type:



Description

sequence-name
Names the sequence. The name, including the implicit or explicit qualifiers, must not identify an existing sequence at the current server, including the sequence names that are generated by DB2.

The schema name must not begin with 'SYS' unless the schema name is 'SYSADM'.

AS *data-type*
Specifies the data type to be used for the sequence value. The data type can be any exact numeric data type (SMALLINT, INTEGER, BIGINT, or DECIMAL with a scale of zero), or a user-defined distinct type for which the source type is an exact numeric data type with a scale of zero. The default, when AS is not specified, is INTEGER. If DECIMAL is specified, the default is DECIMAL(5,0).

START WITH *numeric-constant*
Specifies the first value for the sequence. The value can be any positive or negative value that could be assigned to the a column of the data type that is associated with the sequence without non-zero digits existing to the right of the decimal point.

If the START WITH clause is not explicitly specified with a value, the default is the MINVALUE for ascending sequences and MAXVALUE for descending sequences.

This value is not necessarily the value that a sequence would cycle to after reaching the maximum or minimum value of the sequence. The START WITH clause can be used to start a sequence outside the range that is used for cycles. The range used for cycles is defined by MINVALUE and MAXVALUE.

INCREMENT BY *numeric-constant*
Specifies the interval between consecutive values of the sequence. The value can be any positive or negative value (including 0) that could be assigned to a column of the data type that is associated with the sequence without any non-zero digits existing to the right of the decimal point. The default is 1.

If INCREMENT BY is positive, the sequence ascends. If INCREMENT BY is negative, the sequence descends. If INCREMENT is 0, the sequence is treated as an ascending sequence.

The absolute value of INCREMENT BY can be greater than the difference between MAXVALUE and MINVALUE.

MINVALUE or NO MINVALUE
Specifies the minimum value at which a descending sequence either cycles or stops generating values or an ascending sequence cycles to after reaching the maximum value. The default is NO MINVALUE.

MINVALUE *numeric-constant*
Specifies the minimum end of the range of values for the sequence. The last value that is generated for a cycle of a descending sequence will be equal to or greater than this value. MINVALUE is the value to which an ascending sequence cycles to after reaching the maximum value.

The value can be any positive or negative value that could be assigned to the a column of the data type that is associated with the sequence without non-zero digits existing to the right of the decimal point. The value must be less than or equal to the maximum value.

For the effects of defining MINVALUE and MAXVALUE with the same value, see Defining a constant sequence.

NO MINVALUE
Specifies that the minimum end point of the range of values for the sequence has not been specified explicitly. In such a case, the default value for MINVALUE becomes one of the following:

- For an ascending sequence, the value is the START WITH value or 1 if START WITH is not specified.
- For a descending sequence, the value is the minimum value of the data type that is associated with the sequence.

MAXVALUE or NO MAXVALUE
Specifies the maximum value at which an ascending sequence either cycles or stops generating values or an descending sequence cycles to after reaching the minimum value. The default is NO MAXVALUE.

MAXVALUE *numeric-constant*
Specifies the maximum end of the range of values for the sequence. The last value that is generated for a cycle of an ascending sequence will be less than or equal to this value. MAXVALUE is the value to which a descending sequence cycles to after reaching the minimum value.

The value can be any positive or negative value that could be assigned to the a column of the data type that is associated with the sequence without non-zero digits existing to the right of the decimal point. The value must be greater than or equal to the minimum value.

For the effects of defining MAXVALUE and MINVALUE with the same value, see Defining a constant sequence.

NO MAXVALUE
Specifies the maximum end point of the range of values for the sequence has not been specified explicitly. In such a case, the default value for MAXVALUE becomes one of the following:

- For an ascending sequence, the value is the maximum value of the data type that is associated with the sequence.
- For a descending sequence, the value is the START WITH value or -1 if START WITH is not specified.

CYCLE or NO CYCLE
Specifies whether or not the sequence should continue to generate values after reaching either its maximum or minimum value. The boundary of the sequence can be reached either with the next value landing exactly on the boundary condition or by overshooting it. The default is NO CYCLE.

CYCLE
Specifies that the sequence continue to generate values after either the maximum or minimum value has been reached. If this option is used, after an ascending sequence reaches its maximum value, it generates its minimum value. After a descending sequence reaches its minimum value, it generates its maximum value. The maximum and minimum values for the sequence defined by the MINVALUE and MAXVALUE options determine the range that is used for cycling.

When CYCLE is in effect, duplicate values can be generated by the sequence. When a sequence is defined with CYCLE, any application conversion tools for converting applications from other vendor platforms to DB2 should also explicitly specify MINVALUE, MAXVALUE, and START WITH values.

NO CYCLE
Specifies that the sequence cannot generate more values once the maximum or minimum value for the sequence has been reached. The NO CYCLE option (the default) can be altered to CYCLE at any time during the life of the sequence.

When the next value is being generated for a sequence if the maximum value (for an ascending sequence) or the minimum value (for a descending sequence) of the logical range of the sequence is exceeded and the NO CYCLE option is in effect, an error occurs.

CACHE or NO CACHE
Specifies whether or not to keep some preallocated values in memory for faster access. This is a performance and tuning option.

CACHE *integer-constant*
Specifies the maximum number of values of the sequence that DB2 can preallocate and keep in memory. Preallocating values in the cache reduces synchronous I/O when values are generated for the sequence. The actual number of values that DB2 caches is always the lesser of the number in effect for the CACHE option and the number of remaining values within the logical range. Thus, the CACHE value is essentially an upper limit for the size of the cache.

In the event the system is shut down (either normally or through a system failure), all cached sequence values that have not been used in committed statements are lost (that is, they will never be used). The value specified for the CACHE option is the maximum number of sequence values that could be lost when the system is shut down.

The minimum value is 2. The default is CACHE 20.

In a data sharing environment, you can use the CACHE and NO ORDER options to allow multiple DB2 members to cache sequence values simultaneously.

NO CACHE
Specifies that values of the sequence are not to be preallocated. This option ensures that there is not a loss of values in the case of a system failure. When NO CACHE is specified, the values of the sequence are not stored in the cache. In this case, every request for a new value for the sequence results in synchronous I/O.

ORDER or NO ORDER
Specifies whether the sequence numbers must be generated in order of request. The default is NO ORDER.

ORDER
Specifies that the sequence numbers are generated in order of request. Specifying ORDER might disable the caching of values. There is no guarantee that values are assigned in order across the entire server unless NO CACHE is also specified. ORDER applies only to a single-application process.

NO ORDER
Specifies that the sequence numbers do not need to be generated in order of request.

In a data sharing environment, if the CACHE and NO ORDER options are in effect, multiple caches can be active simultaneously, and the requests for next value assignments from different DB2 members might not result in the assignment of values in strict numeric order. For example, if members DB2A and DB2B are using the same sequence, and DB2A gets the cache values 1 to 20 and DB2B gets the cache values 21 to 40, the actual order of values assigned would be 1,21,2 if DB2A requested for next value first, then DB2B requested, and then DB2A again requested. Therefore, to guarantee that sequence

numbers are generated in strict numeric order among multiple DB2 members using the same sequence concurrently, specify the ORDER option.

Notes

Owner privileges: The owner is authorized to change (ALTER privilege) or use (USAGE privilege) the sequence and grant others these privileges. See "GRANT (sequence privileges)" on page 1714. For more information about ownership of the object see "Authorization, privileges, permissions, masks, and object ownership" on page 70.

Relationship of MINVALUE and MAXVALUE: MINVALUE must not be greater than MAXVALUE. Although MINVALUE is typically less than MAXVALUE, MINVALUE can equal MAXVALUE. If START WITH were the same value as MINVALUE and MAXVALUE, the sequence would be constant. The request for the next value in a constant sequence appears to have no effect because all of the values that are generated by the sequence are in fact the same value.

Defining sequences that cycle: When you define a sequence, you can choose to have it cycle automatically or not when the maximum or minimum value for the sequence has been reached.

- Implicitly or explicitly defining a sequence with NO CYCLE causes the sequence to not cycle automatically after the boundary is reached. However, you can use the ALTER SEQUENCE statement to cycle the sequence manually. ALTER SEQUENCE allows you to restart or extend the sequence, which causes sequence values to continue to be generated.
- Explicitly defining a sequence with CYCLE causes the sequence to cycle automatically after the boundary is reached. Sequence values continue to be generated after the sequence cycles.

  When a sequence is defined to cycle automatically, the maximum or minimum value that is generated for a sequence might not be the actual MAXVALUE or MINVALUE value that is specified if the increment is a value other than 1 or -1. For example, the sequence defined with START WITH=1, INCREMENT=2, MAXVALUE=10 will generate a maximum value of 9, and will not generate the value 10.

  When a sequence is defined with CYCLE, any application conversion tools (for converting applications from other vendor platforms to DB2) should also explicitly specify MINVALUE, MAXVALUE, and START WITH.

Defining a constant sequence: You can define a sequence such that it always returns the same (or a constant) value. To create a constant sequence, use either of these techniques when defining the sequence:

- Specify an INCREMENT value of zero and a START WITH value that does not exceed MAXVALUE.
- Specify the same value for START WITH, MINVALUE, and MAXVALUE, and specify CYCLE.

A constant sequence can be used as a numeric global variable. You can use ALTER SEQUENCE to adjust the values that are generated for a constant sequence.

Consumed values of a sequence: After DB2 generates a value for a sequence, that value can be said to be "consumed" regardless of whether or not that value is used by the application or not. The value is not reused within the current cycle. A consumed value might not be used when the statement that caused the value to be

generated fails for some reason or is rolled back after the value was generated. Generated but unused values can constitute gaps in a sequence.

Gaps in a sequence: Consecutive values in a sequence differ by the constant INCREMENT BY value specified for the sequence. However, gaps can occur in the values that are assigned to a sequence object by DB2.

The following situations are some examples of how gaps can be introduced in the sequence values:

- A transaction has advanced the sequence and then rolls back.
- The SQL statement leading to the generation of the next value fails after the value was generated.
- The NEXT VALUE expression is used in the SELECT statement of a cursor in a DRDA environment where the client uses block-fetch and not all retrieved rows are fetched by the application.
- The sequence is altered and then the alteration is rolled back.
- The sequence (or an identity column table) is dropped and then the drop is rolled back.
- The SYSIBM.SYSSEQ table space is stopped or closed for any reason (including when DSMAX is reached)
- The DB2 subsystem is stopped or goes down

Values of such gaps are not available for the current cycle, unless the sequence is altered and restarted in a specific way to make them available.

A sequence is incremented independently of a transaction. Thus, a given transaction increments the sequence two times might see a gap in the two numbers that it receives if other transactions concurrently increment the same sequence. Most applications can tolerate these instances as these are not really gaps.

Duplicate sequence values: It is possible the duplicate values can be generated for a sequence. Duplicate values are most likely to occur when a sequence is defined with the CYCLE option, is defined as a constant sequence, or is altered. For example, the following situations could cause duplicate sequence values:

- A sequence is defined with the attributes START WITH=2, INCREMENT BY 2, MINVALUE=2, MAXVALUE=10, and CYCLE.
- The ALTER SEQUENCE statement is used to restart the sequence with a value that has already been generated.
- The ALTER SEQUENCE statement is used to reverse the ascending direction of a sequence by changing the INCREMENT BY value from a positive to a negative.

Using sequences: A sequence can be referenced using a *sequence-reference*. A sequence reference can appear in most places that an expression can appear. A sequence reference can specify whether the value to be returned is a newly generated value or the previously generated value. A NEXT VALUE sequence expression is used to generate a new value. A PREVIOUS VALUE sequence expression is used to obtain the last assigned value of a sequence. For more information, see "Sequence reference" on page 291.

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords:

- NOMINVALUE (single key word) as a synonym for NO MINVALUE
- NOMAXVALUE (single key word) as a synonym for NO MAXVALUE

- NOCYCLE (single key word) as a synonym for NO CYCLE
- NOCACHE (single key word) as a synonym for NO CACHE
- NOORDER (single key word) as a synonym for NO ORDER

Examples

Example 1: Create a sequence names "org_seq" that starts at 1 increments by 1, does not cycle, and caches 24 values at a time:

```
CREATE SEQUENCE ORDER_SEQ
   START WITH 1
   INCREMENT BY 1
   NO MAXVALUE
   NO CYCLE
   CACHE 24;
```

INCREMENT 1, NO MAXVALUE, and NO CYCLE are defaults and do not need to be specified.

Example 2: The following example shows how to create and use a sequence named "order_seq" in a table named "orders":

```
CREATE SEQUENCE ORDER_SEQ
   START WITH 1
   INCREMENT BY 1
   NO MAXVALUE
   NO CYCLE
   CACHE 20;
  INSERT INTO ORDERS (ORDERNO, CUSTNO)
    VALUES (NEXT VALUE FOR ORDER_SEQ, 123456);
```

or to update the orders:

```
UPDATE ORDERS
   SET ORDERNO = NEXT VALUE FOR ORDER_SEQ
   WHERE CUSTNO = 123456;
```

Example 3: The following example shows how to use the same sequence number as a unique key value in two separate tables by referencing the sequence number with a NEXT VALUE expression for the first row to generate the sequence value and with a PREVIOUS VALUE expression for the other rows to refer to the sequence value most recently generated.

```
INSERT INTO ORDERS (ORDERNO, CUSTNO)
   VALUES (NEXT VALUE FOR ORDER_SEQ, 123456);
   INSERT INTO LINE_ITEMS (ORDERNO, PARTNO, QUANTITY)
     VALUES (PREVIOUS VALUE FOR ORDER_SEQ, 987654, 100);
```

If NEXT VALUE is invoked in the same statement as the PREVIOUS VALUE, then regardless of their order in the statement, PREVIOUS VALUE returns the previous (unincremented) value and NEXT VALUE returns the next value.

CREATE STOGROUP

The CREATE STOGROUP statement creates a storage group at the current server. Storage from the identified volumes can later be allocated for table spaces and index spaces.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

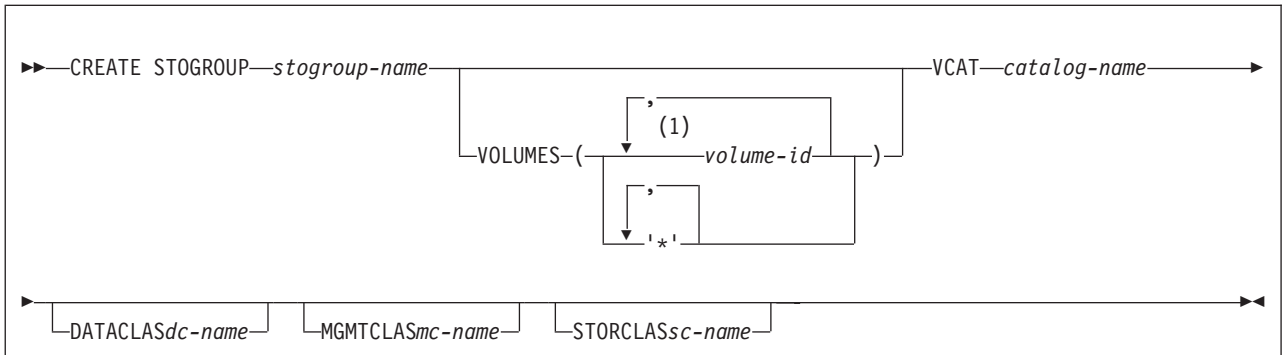
The privilege set that is defined below must include at least one of the following:

- The CREATESG privilege
- SYSADM or SYSCTRL authority

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the of the owner of the plan or package. If the application is bound in a trusted context with the ROLE AS OBJECT OWNER clause specified, a role is the owner. Otherwise, an authorization ID is the owner.

If the statement is dynamically prepared, the privilege set is the privileges that are held by the SQL authorization ID of the process unless the process is within a trusted context and the ROLE AS OBJECT OWNER clause is specified. In that case, the privileges set is the privileges that are held by the role that is associated with the primary authorization ID of the process.

Syntax



Notes:

1 The same *volume-id* must not be specified more than once.

Description

stogroup-name

Names the storage group. The name must not identify a storage group that exists at the current server.

VOLUMES(*volume-id*,...) or VOLUMES('*',...)
Defines the volumes of the storage group. Each *volume-id* is a volume serial number of a storage volume. The volume serial number can have a maximum of six characters and is specified as an identifier or a string constant.

If the data set that is associated with the storage group is not managed by Storage Management Subsystem (SMS), VOLUMES must be specified. Asterisks are recognized only by SMS. SMS usage is recommended, rather than using DB2 to allocate data to specific volumes. Having DB2 select the volume requires non-SMS usage or assigning an SMS Storage Class with guaranteed space. However, because guaranteed space reduces the benefits of SMS allocation, it is not recommended. If one or more of the DATACLAS, MGMTCLAS, or STORCLAS clauses are specified, VOLUMES can be omitted. If the VOLUMES clause is omitted, the volume selection is controlled by SMS.

If you do choose to use specific volume assignments, additional manual space management must be performed. Free space must be managed for each individual volume to prevent failures during the initial allocation and extension. This process generally requires more time for space management and results in more space shortages. Guaranteed space should be used only where the space needs are relatively small and do not change.

VCAT *catalog-name*
Identifies the integrated catalog facility catalog for the storage group. You must specify an alias[31] if the name of the integrated catalog facility catalog is longer than 8 characters.

The designated catalog is the one in which entries are placed for the data sets created by DB2 with the aid of the storage group. These are linear VSAM data sets for associated table or index spaces or for their partitions. For each such space or partition, association is made through a USING clause in a CREATE TABLESPACE, CREATE INDEX, ALTER TABLESPACE, or ALTER INDEX statement. For more on the association, see the descriptions of those statements in this chapter.

Conventions for data set names are given in *DB2 Administration Guide*. *catalog-name* is the first qualifier for each data set name.

One or more DB2 subsystems could share integrated catalog facility catalogs with the current server. To avoid the chance of having one of those subsystems attempt to assign the same name to different data sets, select a value for *catalog-name* that is not used by the other DB2 subsystems.

DATACLAS *dc-name*
Identifies the name of the SMS data class to associate with the DB2 storage group. The SMS data class name must be from 1-8 characters in length. The SMS storage administrator defines the data class that can be used. DATACLAS must not be specified more than one time.

MGMTCLAS *mc-name*
Identifies the name of the SMS management class to associate with the DB2 storage group. The SMS management class name must be from 1-8 characters in length. The SMS storage administrator defines the management class that can be used. MGMTCLAS must not be specified more than one time.

STORCLAS *sc-name*
Identifies the name of the SMS storage class to associate with the DB2 storage group. The SMS storage class name must be from 1-8 characters in length. The

31. The alias of an integrated catalog facility catalog.

SMS storage administrator defines the storage class that can be used. STORCLAS must not be specified more than one time.

Notes

Device types: When the storage group is used at run time, an error can occur if the volumes in the storage group are of different device types, or if a volume is not available to z/OS for dynamic allocation of data sets.

When a storage group is used to extend a data set, all volumes in the storage group must be of the same device type as the volumes used when the data set was defined. Otherwise, an extend failure occurs if an attempt is made to extend the data set.

Number of volumes: There is no specific limit on the number of volumes that can be defined for a storage group. However, the maximum number of volumes that can be managed for a storage group is 133.

z/OS imposes a limit on the number of volumes that can be allocated per data set (currently, 59 volumes). For the latest information on that restriction, see *z/OS DFSMS Access Method Services for Catalogs*.

Storage group owner: If the statement is embedded in an application program, the owner of the plan or package is the owner of the storage group. If the statement is dynamically prepared, the SQL authorization ID of the process is the owner of the storage group. The owner has the privilege of altering and dropping the storage group.

Specifying volume IDs: A new storage group must have either specific volume IDs or non-specific volume IDs. You cannot create a storage group that contains a mixture of specific and non-specific volume IDs.

Verifying the existence of volumes and classes: When processing the VOLUMES, DATACLAS, MGMTCLAS, or STORCLAS clauses, DB2 does not check the existence of the volumes or classes or determine the types of devices that are identified or if SMS is active. Later, when the storage group allocates data sets, the list of volumes is passed in the specified order to Data Facilities (DFSMSdfp). See *DB2 Administration Guide* for more information about creating DB2 storage groups.

Example

Create storage group, DSN8G110, of volumes ABC005 and DEF008. DSNCAT is the integrated catalog facility catalog name.

```
CREATE STOGROUP DSN8G110
  VOLUMES (ABC005,DEF008)
  VCAT DSNCAT;
```

CREATE SYNONYM

The CREATE SYNONYM statement defines a synonym for a table or view at the current server.

Important: Synonyms behave differently with DB2 for z/OS than with the other DB2 family products. Synonyms are not recommended for use when writing new SQL statements or when creating portable applications. Use aliases instead.

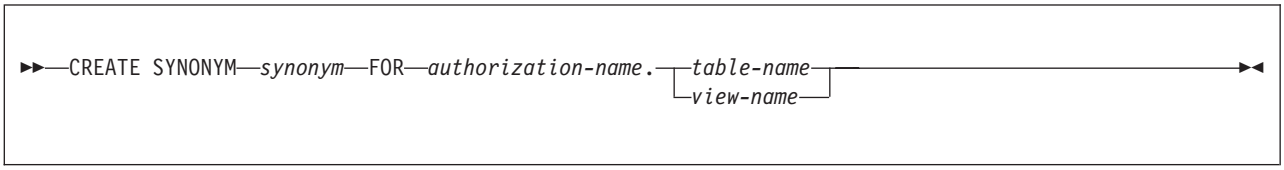
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified. The statement cannot be processed in a trusted context that is defined with a role as the object owner.

Authorization

None required.

Syntax



Description

synonym
Names the synonym. The name must not identify a synonym, table, view, or alias that exists at the current server and that is owned by the owner of the synonym that is being created and must not identify a table that exists in the SYSIBM.SYSPENDINGOBJECTS catalog table. The unqualified name must not be the same as an existing synonym.

FOR *authorization-name.table-name* **or** *authorization-name.view-name*
Identifies the object to which the synonym applies. The name must consist of two parts and must identify a table, view, or alias that exists at the current server. If a table is identified, it must not be an auxiliary table or a declared temporary table. If an alias is identified, it must be an alias for a table or view at the current server and the synonym is defined for that table or view. The name must not identify a table that was implicitly created for an XML column.

Notes

Owner privileges: There are no specific privileges on a synonym. For more information about ownership of an object, see "Authorization, privileges, permissions, masks, and object ownership" on page 70.

The authorization ID recorded as the owner of a synonym is the only authorization ID for which the synonym is defined.

If an alias is used to denote the table or view, the name of that table or view, not the alias, is recorded in the catalog as the definition of the synonym. That severs the connection between the synonym and alias, and even if the alias is dropped and redefined, the synonym is still in effect and names the original table or view.

Example

Define DEPT as a synonym for the table DSN8B10.DEPT.

```
CREATE SYNONYM DEPT
  FOR DSN8B10.DEPT;
```

CREATE TABLE

The CREATE TABLE statement defines a table. The definition must include its name and the names and attributes of its columns. The definition can include other attributes of the table, such as its primary key and its table space.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set that is defined below must include at least one of the following:

- The CREATETAB privilege for the database explicitly specified by the IN clause.

  If the IN clause is not specified, the CREATETAB privilege on database DSNDB04 is required.
- DBADM, DBCTRL, or DBMAINT authority for the database explicitly specified by the IN clause. If the IN clause is not specified, DBADM, DBCTRL, or DBMAINT authority for database DSNDB04 is required.
- SYSADM or SYSCTRL authority
- System DBADM

If the table space is created implicitly, the privilege set that is defined below must include at least one of the following:

- The CREATETS privilege for the database explicitly specified by the IN clause.

  If the IN clause is not specified, the CREATETS privilege on database DSNDB04 is required.
- DBADM, DBCTRL, or DBMAINT authority for the database explicitly specified by the IN clause. If the IN clause is not specified, DBADM, DBCTRL, or DBMAINT authority for database DSNDB04 is required.
- SYSADM or SYSCTRL authority

The privilege set must also have the USE privilege for the default buffer pool and default storage group of the database if the database is specified in the IN clause.

For tables that are created in an implicit database, the database authority must be held on DSNDB04.

Additional privileges might be required in the following conditions:

- The clause IN, LIKE or FOREIGN KEY is specified.
- The data type of a column is a distinct type.
- The table space is implicitly created.
- A *fullselect* is specified.
- A column is defined as a security label column.

See the description of the appropriate clauses for details about these privileges.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package.

If the application is bound in a trusted context with the ROLE AS OBJECT OWNER clause specified:

- A role is the owner of the table that is being created
- The privilege set is the set of privileges that are held by that role
- The schema qualifier (implicit or explicit) must be the same as the role, unless the role has the CREATEIN privilege on the schema, or SYSADM, SYSCTRL, or System DBADM authority

Otherwise, an authorization ID is the owner of the plan or package, and the following rules apply:

- If the privilege set lacks the CREATEIN privilege on the schema, SYSADM authority, SYSCTRL authority, and System DBADM authority, the schema qualifier (implicit or explicit) must be the same as the authorization ID of the owner of the plan or package.
- If the privilege set lacks SYSADM authority, SYSCTRL authority, and System DBADM authority, and the table is explicitly qualified, the authorization ID that is the same as the schema name must have all the necessary privileges to create the table, and that authorization ID is the owner of the table. Otherwise, the authorization ID of the owner of the plan or package must have all the necessary privileges to create the table, and that authorization ID is the owner of the table.
- If the privilege set includes SYSADM authority, SYSCTRL authority, or System DBADM authority, the schema qualifier (implicit or explicit) can be any schema name. However, if the table is explicitly qualified, the authorization ID that is the same as the schema name is the owner of the table. Otherwise, the authorization ID of the owner of the plan or package is the owner of the table.

If the statement is dynamically prepared, the privilege set is the privileges that are held by the SQL authorization ID of the process unless the process is within a trusted context and the ROLE AS OBJECT OWNER clause is in effect. When ROLE AS OBJECT OWNER is in effect, the privileges set is the privileges that are held by the role that is associated with the primary authorization ID of the process, and the owner of the table is that role. The schema qualifier (implicit or explicit) must be the same as that role, unless the role has CREATEIN privilege on the schema, or SYSADM authority, SYSCTRL authority, or System DBADM authority.

For the case where the SQL authorization ID of the process holds the privileges, the following rules apply:

- If the privilege set lacks CREATEIN privilege on the schema, SYSADM authority, SYSCTRL authority, and System DBADM authority, the schema qualifier must be the same as one of the authorization IDs of the process.
- If the privilege set lacks SYSADM authority, SYSCTRL authority, and System DBADM authority, and the table is explicitly qualified, then the authorization ID that is the same as the schema name must have all the necessary privileges to create the table, and that authorization ID is the owner of the table. Otherwise, the SQL authorization ID of the process must include all privileges that are needed to create the table, and that authorization ID is the owner of the table.
- If the privilege set includes SYSADM authority, SYSCTRL authority, or System DBADM authority, the schema qualifier can be any schema name. However, if the table is explicitly qualified, then the authorization ID that is the same as the schema name is the owner of the table. Otherwise, the SQL authorization ID of the process is the owner of the table.

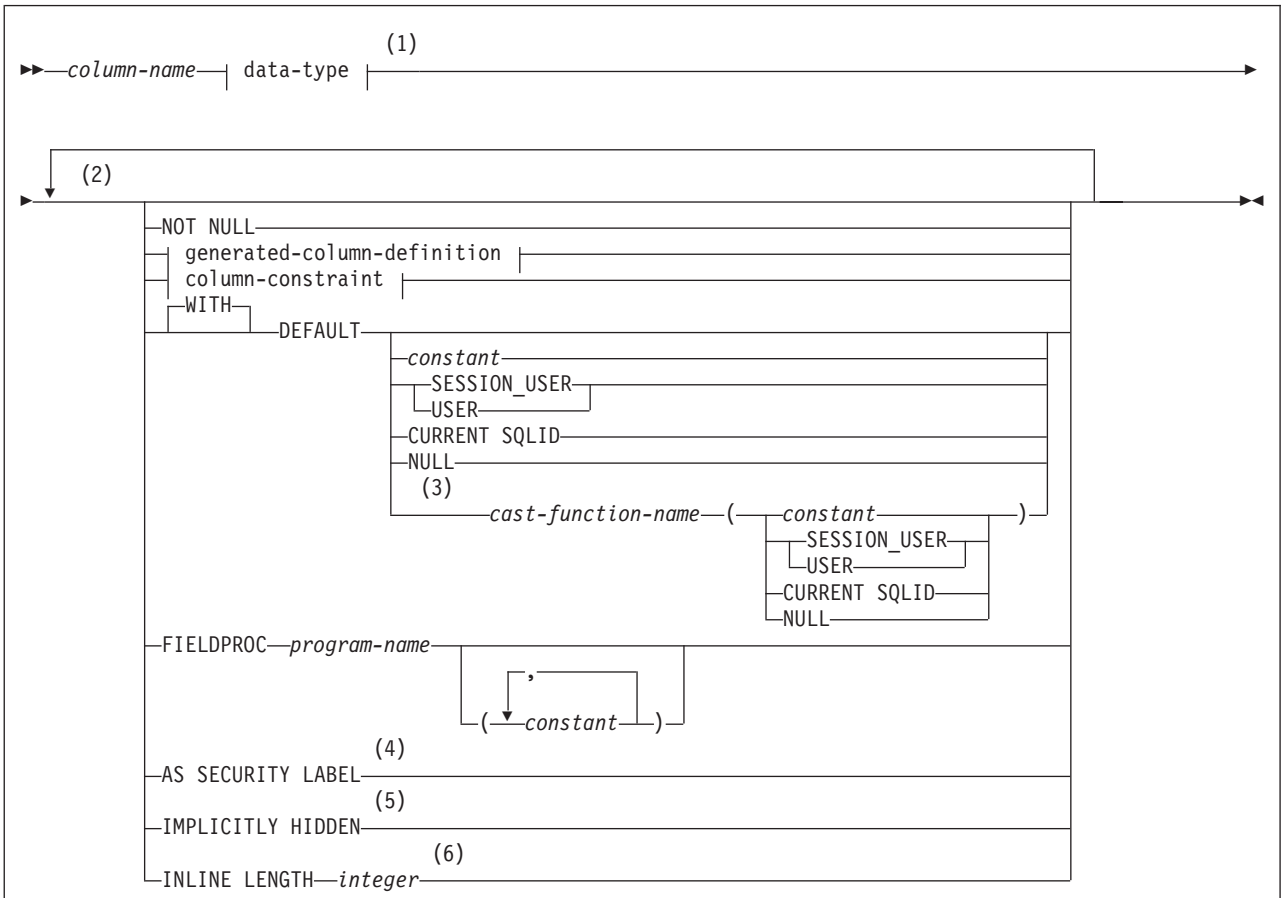
Syntax


CREATE TABLE
table-name
(
,
column-definition
period-definition
unique-constraint
referential-constraint
check-constraint
)
LIKE
table-name
view-name
copy-options
as-result-table
copy-options
materialized-query-definition
(1)
IN
database-name.
table-space-name
IN DATABASE
database-name
partitioning-clause
organization-clause
EDITPROC
program-name
WITH ROW ATTRIBUTES
WITHOUT ROW ATTRIBUTES
VALIDPROC
program-name
AUDIT NONE
AUDIT CHANGES
AUDIT ALL
OBID
integer
DATA CAPTURE NONE
DATA CAPTURE CHANGES
WITH RESTRICT ON DROP
CCSID
ASCII
EBCDIC
UNICODE
NOT VOLATILE
CARDINALITY
VOLATILE
CARDINALITY
LOGGED
NOT LOGGED
COMPRESS NO
COMPRESS YES
APPEND
NO
YES
DSSIZE
integer
G
BUFFERPOOL
bpname
MEMBER CLUSTER
TRACKMOD YES
TRACKMOD NO

Notes:

1 The same clause must not be specified more than once.

column-definition:



Notes:

1 Data type is optional if *as-row-change-timestamp-clause* is specified

2 The same clause must not be specified more than one time.

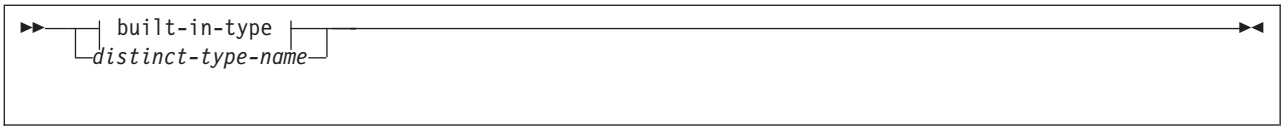
3 This form of the DEFAULT value can only be used with columns that are defined as a distinct type.

4 AS SECURITY LABEL can be specified only for a CHAR(8) data type and requires that the NOT NULL and WITH DEFAULT clauses be specified.

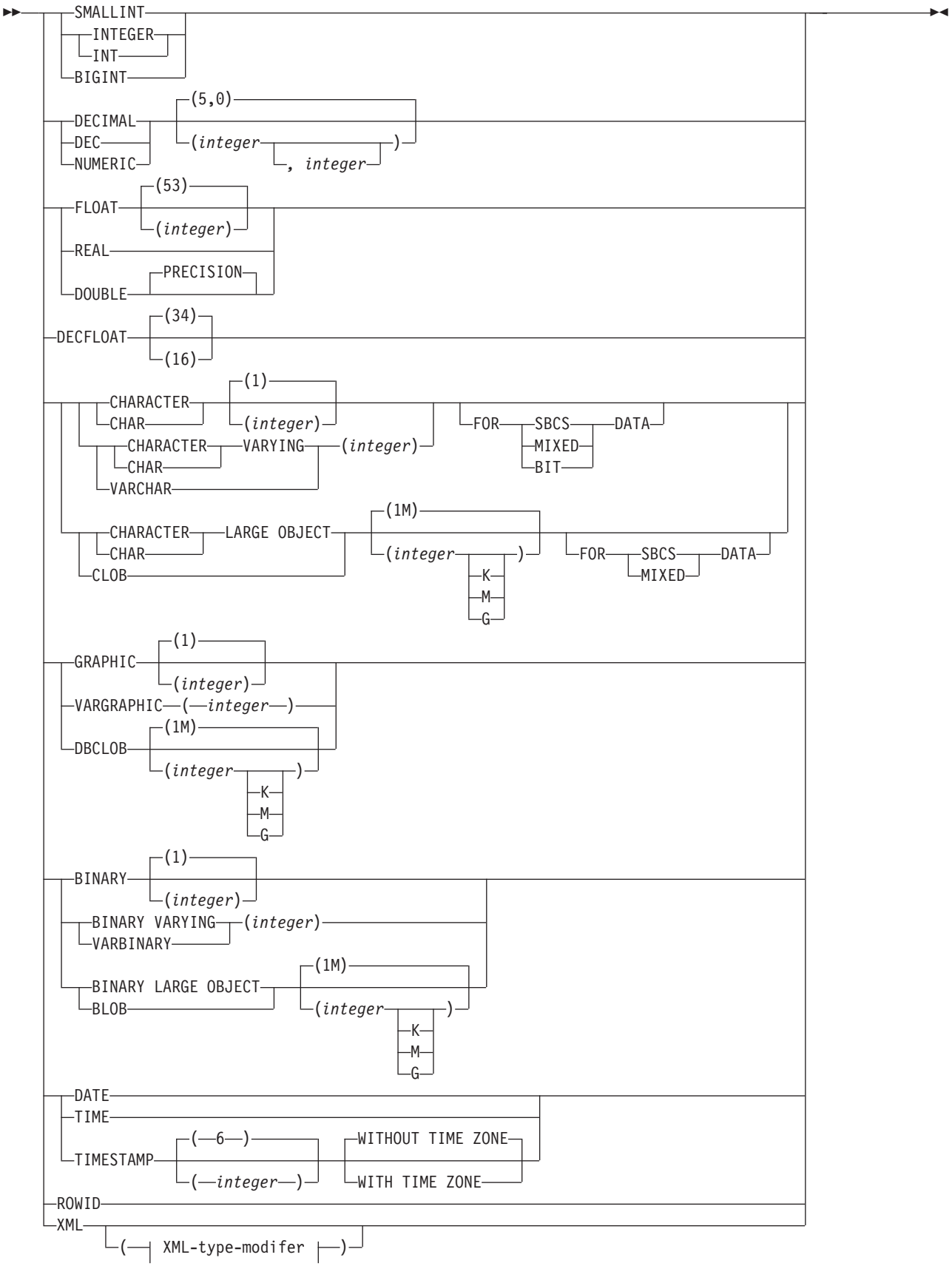
5 IMPLICITLY HIDDEN must not be specified for a column defined as a ROWID, or a distinct type that is based on a ROWID.

6 INLINE LENGTH only applies to a column with a LOB data type or a distinct type that is based on a LOB data type.

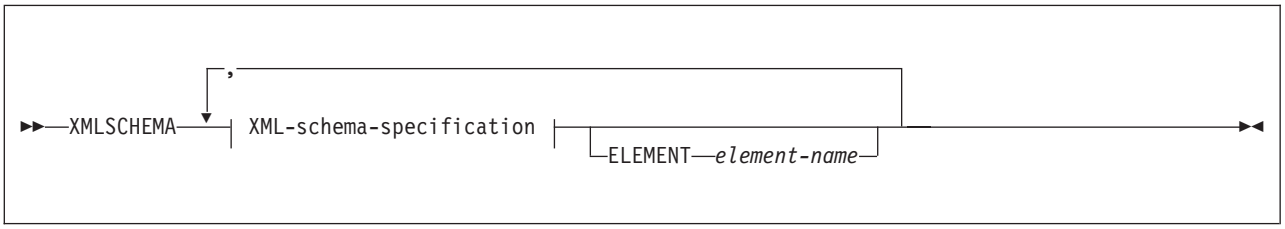
data-type:



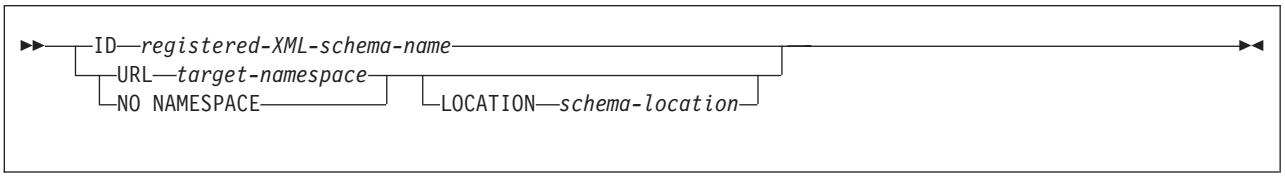
built-in-type:


SMALLINT
INTEGER
INT
BIGINT
DECIMAL
DEC
NUMERIC
(5,0)
(integer)
, integer
FLOAT
(53)
(integer)
REAL
DOUBLE
PRECISION
DECFLOAT
(34)
(16)
CHARACTER
CHAR
(1)
(integer)
CHARACTER
CHAR
VARCHAR
VARYING
(integer)
FOR
SBCS
MIXED
BIT
DATA
CHARACTER
CHAR
CLOB
LARGE OBJECT
(1M)
(integer)
K
M
G
FOR
SBCS
MIXED
DATA
GRAPHIC
(1)
(integer)
VARGRAPHIC
(integer)
DBCLOB
(1M)
(integer)
K
M
G
BINARY
(1)
(integer)
BINARY VARYING
VARBINARY
(integer)
BINARY LARGE OBJECT
BLOB
(1M)
(integer)
K
M
G
DATE
TIME
TIMESTAMP
(6)
(integer)
WITHOUT TIME ZONE
WITH TIME ZONE
ROWID
XML
(XML-type-modifer)

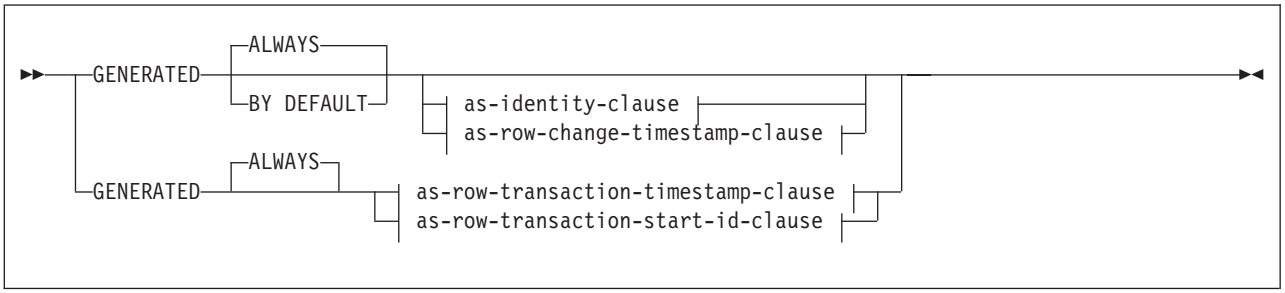
XML-type-modifer:



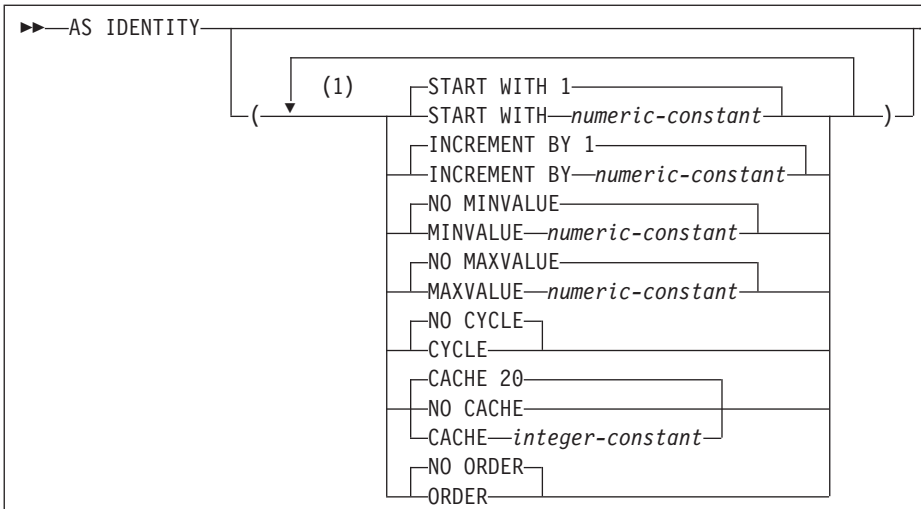
XML-schema-specification:



generated-column-definition:



as-identity-clause:

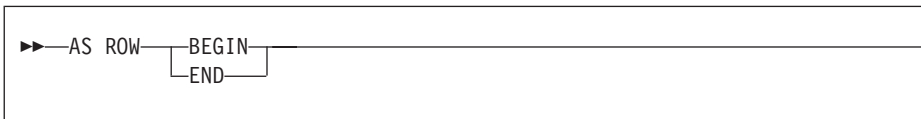


Notes:

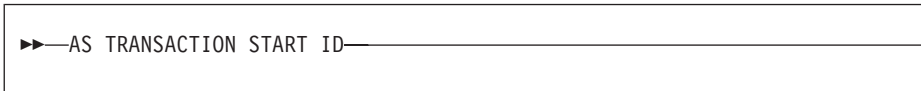
1 Separator commas can be specified between attributes when an identity column is defined.

as-row-change-timestamp-clause:

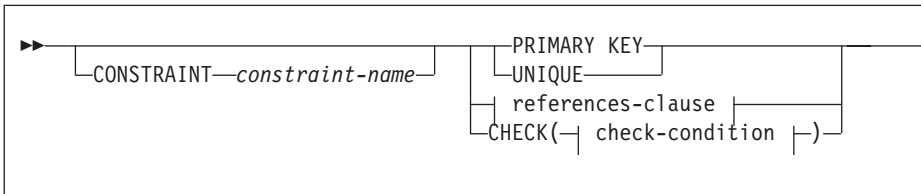
as-row-transaction-timestamp-clause:



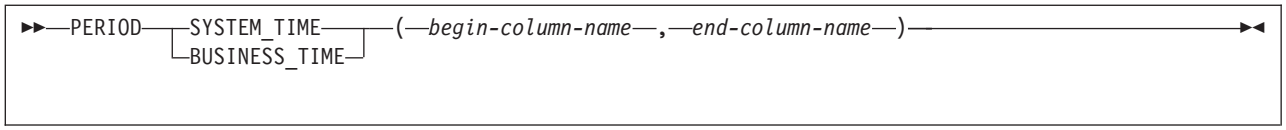
as-row-transaction-start-id-clause:



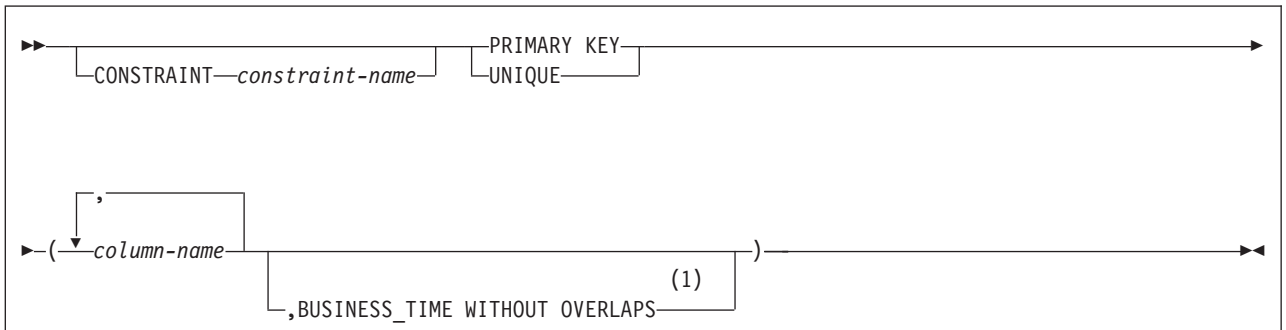
column-constraint:



period-definition:



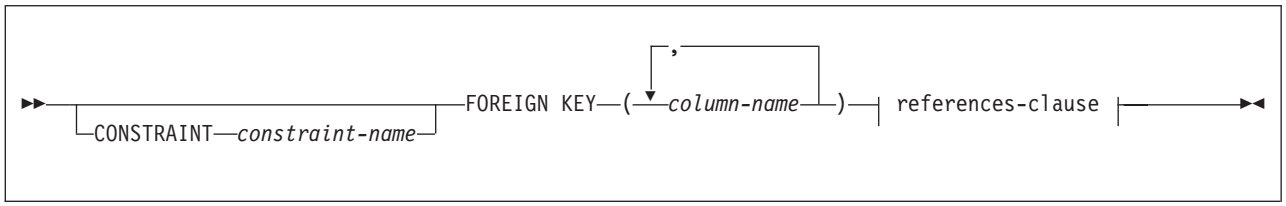
unique-constraint:



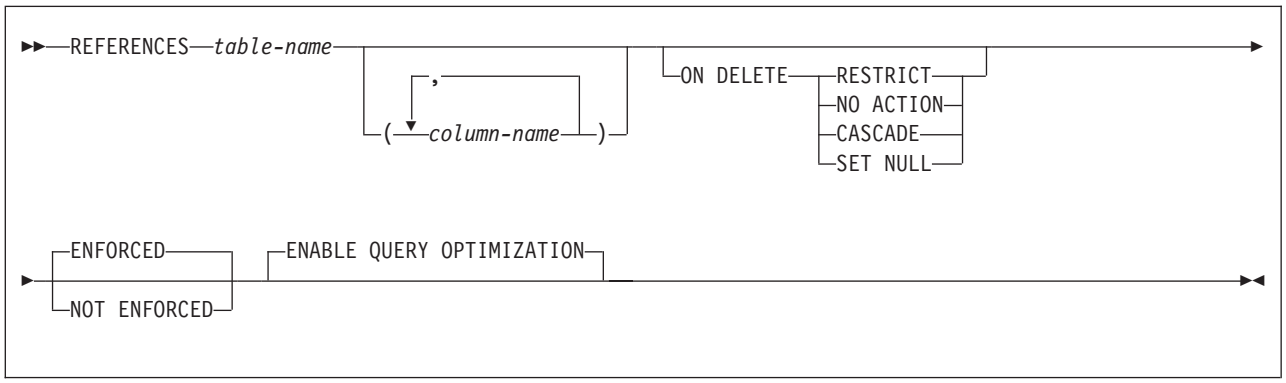
Notes:

1 If **BUSINESS_TIME WITHOUT OVERLAPS** is specified, the BUSINESS_TIME period will not overlap in time periods for the same *column-name* values.

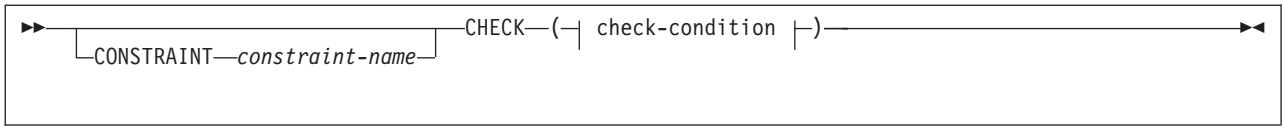
referential-constraint:



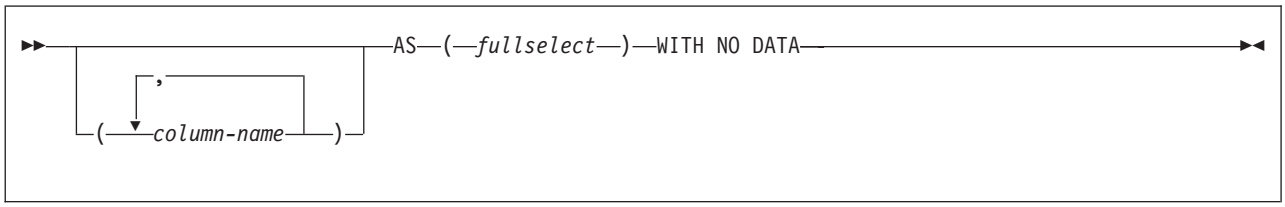
references-clause:



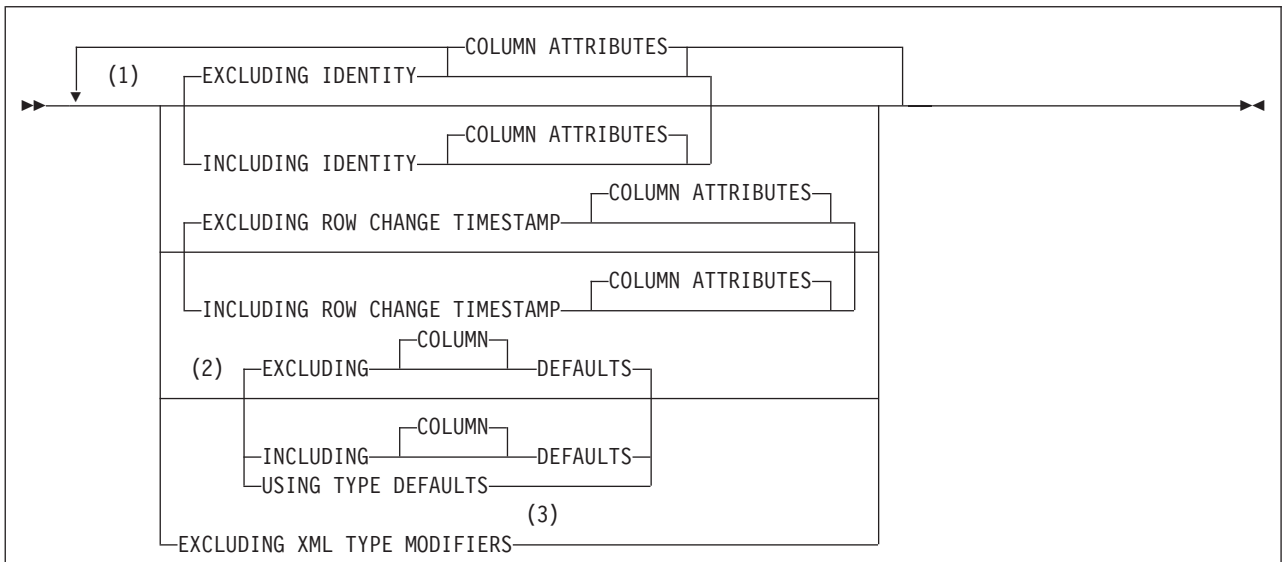
check-constraint:



as-result-table:



copy-options:



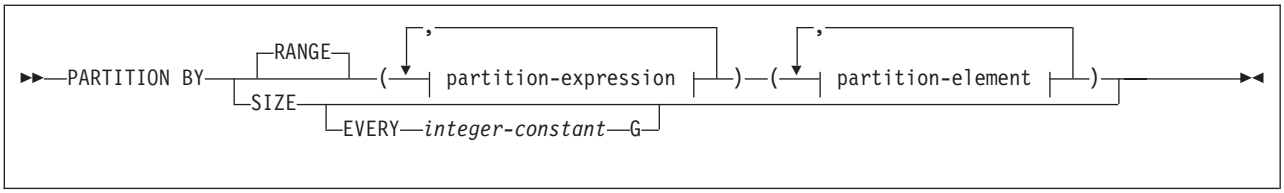
Notes:

1 These clauses can be specified in any order and must not be specified more than one time.

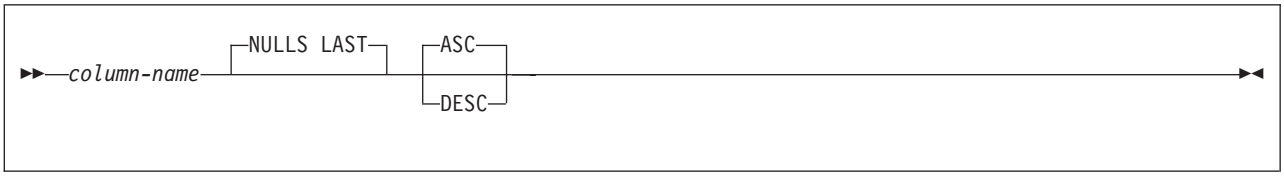
2 EXCLUDING COLUMN DEFAULTS, INCLUDING COLUMN DEFAULTS, and USING TYPE DEFAULTS must not be specified with the LIKE clause.

3 EXCLUDING XML TYPE MODIFIERS must be specified with the LIKE clause if the identified table has an XML type modifier and none of the XML columns of the new table has an XML type modifier. EXCLUDING XML TYPE MODIFIERS is not supported when a view is identified in a LIKE clause and the view contains XML columns.

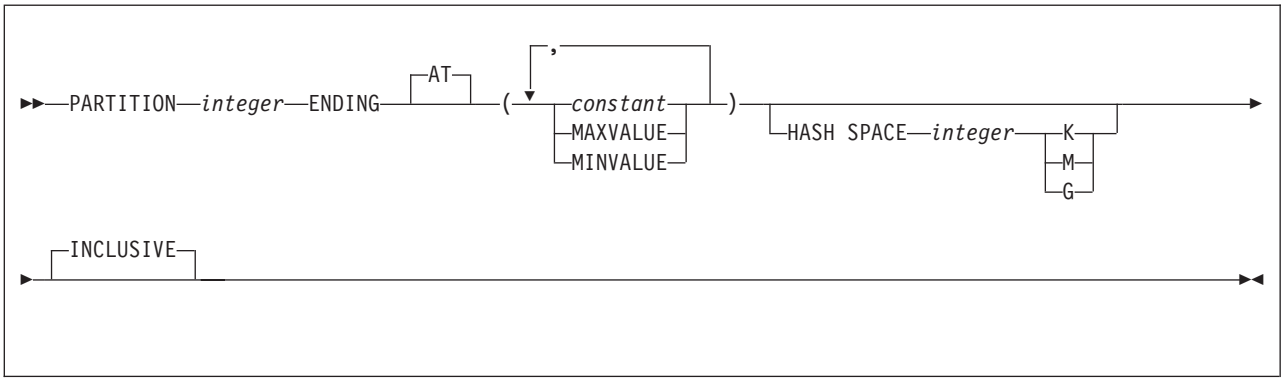
partitioning-clause:



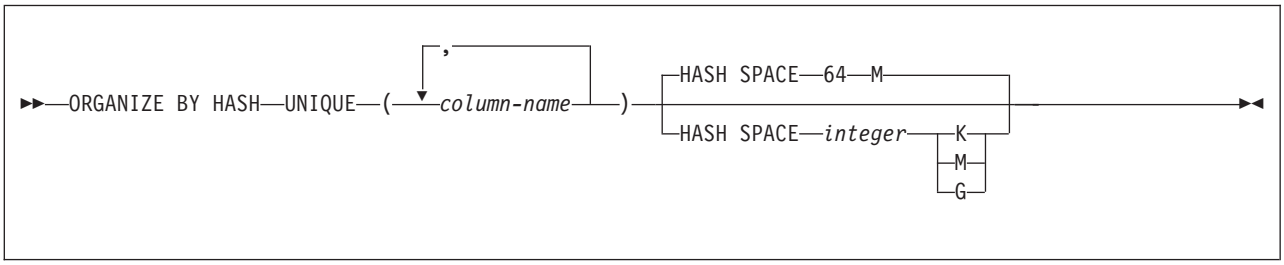
partition-expression:



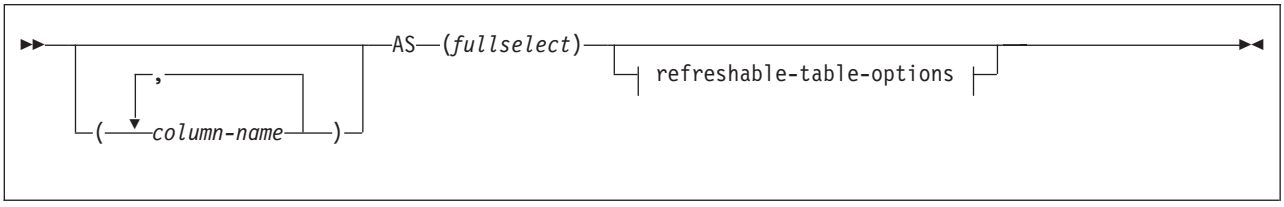
partition-element:



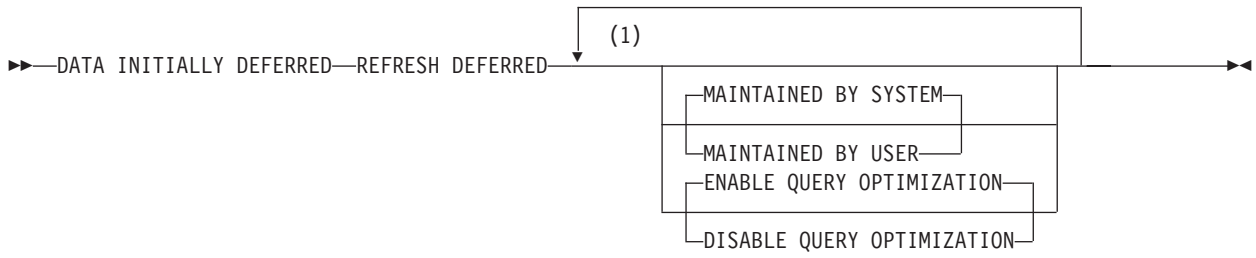
organization-clause



materialized-query-definition



refreshable-table-options:



Notes:

1 The same clause must not be specified more than one time.

Description

table-name
Names the table. The name, including the implicit or explicit qualifier, must not identify a table, view, alias, or synonym that exists at the current server or a table that exists in the SYSIBM.SYSPENDINGOBJECTS catalog table. The unqualified name must not be the same as an existing synonym.

If the name is qualified, the name can be a two-part or three-part name. If a three-part name is used, the first part must match the value of field DB2 LOCATION NAME on installation panel DSNTIPR at the current server. (If the current server is not the local DB2, this name is not necessarily the name in the CURRENT SERVER special register.)

column-name
Names a column of the table. For a dependent table, up to 749 columns can be named. For a table that is not a dependent, this number is 750. Do not qualify *column-name* and do not use the same name for more than one column of the table.

built-in-type
Specifies the data type of the column as one of the following built-in data types, and for character string data types, specifies the subtype. For more information about defining a table with a LOB column (CLOB, BLOB, or DBCLOB), see Creating a table with LOB columns.

SMALLINT
For a small integer.

INTEGER or INT
For a large integer.

BIGINT
For a big integer.

DECIMAL(*integer***,***integer***) or DEC(***integer***,***integer***)**
DECIMAL(*integer***) or DEC(***integer***)**
DECIMAL or DEC
For a decimal number. The first integer is the precision of the number. That is, the total number of digits, which can range from 1 to 31. The second integer is the scale of the number. That is, the number of digits to the right of the decimal point, which can range from 0 to the precision of the number.

You can use DECIMAL(*p*) for DECIMAL(*p*,0) and DECIMAL for DECIMAL(5,0).

You can also use the word NUMERIC instead of DECIMAL. For example, NUMERIC(8) is equivalent to DECIMAL(8). Unlike DECIMAL, NUMERIC has no allowable abbreviation.

DECFLOAT(*integer***)**
For a decimal floating-point number. The value of *integer* must be either 16 or 34 and represents the number of significant digits that can be stored. If *integer* is omitted, the DECFLOAT column will be capable of representing 34 significant digits.

FLOAT(*integer***)**
FLOAT
For a floating-point number. If *integer* is between 1 and 21 inclusive, the format is single precision floating-point. If the integer is between 22 and 53 inclusive, the format is double precision floating-point.

You can use DOUBLE PRECISION or FLOAT for FLOAT(53).

REAL
For single precision floating-point.

DOUBLE or DOUBLE PRECISION
For double precision floating-point

CHARACTER(*integer***) or CHAR(***integer***)**
CHARACTER or CHAR
For a fixed-length character string of length *integer*, which can range from 1 to 255. If the length specification is omitted, a length of 1 character is assumed.

VARCHAR(*integer***), CHAR VARYING(***integer***), or CHARACTER VARYING(***integer***)**
For a varying-length character string of maximum length *integer*, which can range from 1 to the maximum record size minus 10 bytes. See Table 119 on page 1448 to determine the maximum record size.

FOR *subtype* **DATA**
Specifies a subtype for a character string column, which is a column with a data type of CHAR, VARCHAR, or CLOB. Do not use the FOR *subtype* DATA clause with columns of any other data type (including any distinct type). *subtype* can be one of the following:

SBCS
Column holds single-byte data.

MIXED
Column holds mixed data. Do not specify MIXED if the value of field MIXED DATA on installation panel DSNTIPF is NO unless the CCSID UNICODE clause is also specified, or the table is being created in a Unicode table space or database.

BIT
Column holds BIT data. Do not specify BIT for a CLOB column.

Only character strings are valid when subtype is BIT.

If you do not specify the FOR clause, the column is defined with a default subtype. For ASCII or EBCDIC data:

- The default is SBCS when the value of field MIXED DATA on installation panel DSNTIPF is NO.

- The default is MIXED when the value is YES.

For Unicode data, the default subtype is MIXED.

A security label column is always considered SBCS data, regardless of the encoding scheme of the table.

CCSID 1208
Specifies that the column is a Unicode column encoded in UTF8. This clause must not be specified for an ASCII or Unicode table.

CLOB(*integer* **[K|M|G]), CHAR LARGE OBJECT(***integer* **[K|M|G]), or CHARACTER LARGE OBJECT(***integer* **[K|M|G])**
CLOB, CHAR LARGE OBJECT, or CHARACTER LARGE OBJECT
For a character large object (CLOB) string of the specified maximum length in bytes. The maximum length must be in the range of 1 to 2 147 483 647. A CLOB column has a varying-length. It cannot be referenced in certain contexts regardless of its maximum length. For more information, see "Restrictions using LOBs" on page 97.

When *integer* is not specified, the default length is **1M**. The maximum value that can be specified for *integer* depends on whether a units indicator is also specified as shown in the following list.

integer The maximum value for *integer* is 2 147 483 647. The maximum length of the string is *integer*.

integer **K**
The maximum value for *integer* is 2 097 152. The maximum length is 1024 times *integer*.

integer **M**
The maximum value for *integer* is 2048. The maximum length is 1 048 576 times *integer*.

integer **G**
The maximum value for *integer* is 2. The maximum length is 1 073 741 824 times *integer*.

If you specify a value that evaluates to 2 gigabytes (2 147 483 648), DB2 uses a value that is one byte less, or 2 147 483 647.

GRAPHIC(*integer***)**
GRAPHIC
For a fixed-length graphic string of length *integer*, which can range from 1 to 127. If the length specification is omitted, a length of 1 character is assumed.

VARGRAPHIC(*integer***)**
For a varying-length graphic string of maximum length *integer*, which must range from 1 to *n*/2, where *n* is the maximum row size minus 2 bytes.

CCSID 1200
Specifies that the column is a Unicode column encoded in UTF16. This clause must not be specified for an ASCII or Unicode table.

DBCLOB(*integer* **[K|M|G])**
DBCLOB
For a double-byte character large object (DBCLOB) string of the specified maximum length in double-byte characters. The maximum length must be in the range of 1 through 1 073 741 823. A DBCLOB column has a

varying-length. It cannot be referenced in certain contexts regardless of its maximum length. For more information, see “Restrictions using LOBs” on page 97.

When *integer* is not specified, the default length is **1M**. The meaning of *integer* K|M|G is similar to CLOB. The difference is that the number specified is the number of double-byte characters.

BINARY(*integer***)**
A fixed-length binary string of length *integer*. The *integer* can range from 1 through 255. If the length specification is omitted, a length of 1 byte is assumed.

BINARY VARYING(*integer***) or VARBINARY(***integer***)**
A varying-length binary string of maximum length *integer*, which can range from 1 through 32704. The length is limited by the page size of the table space.

BLOB (*integer* **[K|M|G] or BINARY LARGE OBJECT(***integer* **[K|M|G])**
BLOB or BINARY LARGE OBJECT
For a binary large object (BLOB) string of the specified maximum length in bytes. The maximum length must be in the range of 1 through 2 147 483 647. A BLOB column has a varying-length. It cannot be referenced in certain contexts regardless of its maximum length. For more information, see “Restrictions using LOBs” on page 97.

When *integer* is not specified, the default length is **1M**. The meaning of *integer* K|M|G is the same as for CLOB.

DATE
For a date.

TIME
For a time.

TIMESTAMP(*integer***) WITHOUT TIME ZONE**
For a timestamp. *integer* specifies the optional timestamp precision attribute and must be in the range from 0 to 12. The timestamp precision denotes the number of fractional second digits that are included in the timestamp. The default is 6.

TIMESTAMP(*integer***) WITH TIME ZONE**
For a timestamp with time zone. *integer* specifies the optional timestamp precision attribute and must be in the range from 0 to 12. The timestamp precision denotes the number of fractional second digits that are included in the timestamp. The default is 6.

ROWID
For a row ID type.

A table can have only one ROWID column. The values in a ROWID column are unique for every row in the table and cannot be updated. You must specify NOT NULL with ROWID.

XML
For an XML document. Only well-formed XML documents can be inserted into an XML column.

If the XML column is the first XML column that you create for the table, a BIGINT DOCID column is implicitly created and is used to store a unique document identifier for the XML columns of a row.

XMLSCHEMA
Specifies one or more XML schemas that are used to validate the XML value. The same XML schema can not be specified more than one time.

If the XML value has already been validated, for example, the XML value is the result of the DSN_XMLVALIDATE function or from an XML column with a type modifier, and the XML schema against which the XML value is validated is one of the schemas specified in the *XML-type-modifier*, DB2 accepts the XML value without revalidation.

XML-schema-specification
Specifies one XML schema. The XML schema can be identified by using either the registered XML-schema-name or the schema's target namespace followed by an optional schema location. Any XML schema that is referenced in this clause must be registered in the XML schema repository prior to use.

ID *registered-XML-schema-name*
Identifies an XML schema by using its *registered-XML-schema-name*. The name must uniquely identify an existing XML schema in the XML schema repository at the current server. If no XML schema by this name exists, an error is returned.

The schema qualifier must be SYSXSR.

URI *target-namespace*
Specifies the target namespace URI of the XML schema. The value for the target-namespace URI is a character string constant which is not empty. The URI must be the target namespace of a registered XML schema and, if no LOCATION clause is specified, it must uniquely identify the registered XML schema.

NO NAMESPACE
Specifies that the XML schema has no target namespace. There must be a registered XML schema that has no target namespace. If no LOCATION clause is specified, there must be only one such registered XML schema.

LOCATION *schema-location*
Specifies the XML schema location URI of the XML schema. The value of *schema-location* is a character string constant that is not empty. The schema location URI, combined with the target namespace URI, must identify a registered XML schema.

ELEMENT *element-name*
Specifies the name of the global element declaration. *element-name* must match the local name of the root element node in the instance XML document. The namespace name of the root element node must be the same as the target namespace URI.

distinct-type-name
Specifies the data type of the column is a distinct type (a user-defined data type). The length, precision, and scale of the column are respectively the length, precision, and scale of the source type of the distinct type. The privilege set must implicitly or explicitly include the USAGE privilege on the distinct type.

The encoding scheme of the distinct type must be the same as the encoding scheme of the table. The subtype for the distinct type, if it has the attribute, is the subtype with which the distinct type was created.

If the column is to be used in the definition of the foreign key of a referential constraint, the data type of the corresponding column of the parent key must have the same distinct type.

NOT NULL
Prevents the column from containing null values. Omission of NOT NULL implies that the column can contain null values.

column-constraint
The *column-constraint* of a *column-definition* provides a shorthand method of defining a constraint composed of a single column. Thus, if a *column-constraint* is specified in the definition of column C, the effect is the same as if that constraint were specified as a *unique-constraint*, *referential-constraint*, or *check-constraint* in which C is the only identified column.

CONSTRAINT *constraint-name*
Names the constraint. If a constraint name is not specified, a unique constraint name is generated. If the name is specified, it must be different from the names of any referential, check, primary key, or unique key constraints previously specified on the table.

PRIMARY KEY
Provides a shorthand method of defining a primary key composed of a single column. Thus, if PRIMARY KEY is specified in the definition of column C, the effect is the same as if the PRIMARY KEY(C) clause is specified as a separate clause.

The NOT NULL clause must be specified with this clause. PRIMARY KEY cannot be specified more than one time in a column definition, and must not be specified if the UNIQUE clause is specified in the definition. This clause must also not be specified if the definition is for one of the following types of columns:

- a LOB column
- a ROWID column
- a DECFLOAT column
- a distinct type column that is based on a LOB, ROWID, or DECFLOAT data type
- an XML column
- a row change timestamp column
- a Unicode column in an EBCDIC table

The table is marked as unavailable until its primary index is explicitly created unless the CREATE TABLE statement is processed by the schema processor or the table space that contains the table is implicitly created. In that case, DB2 implicitly creates an index to enforce the uniqueness of the primary key and the table definition is considered complete. (For more information about implicitly created indexes, see Implicitly created indexes.)

UNIQUE
Provides a shorthand method of defining a unique key composed of a single column. Thus, if UNIQUE is specified in the definition of column C, the effect is the same as if the UNIQUE(C) clause is specified as a separate clause.

The NOT NULL clause must be specified with this clause. UNIQUE cannot be specified more than one time in a column definition and must not be

specified if the PRIMARY KEY clause is specified in the column definition or if the definition is for one of the following types of columns:

- a LOB column
- a ROWID column
- a DECFLOAT column
- a distinct type column that is based on a LOB, ROWID, or DECFLOAT data type
- an XML column
- a row change timestamp column
- a Unicode column in an EBCDIC table

The table is marked as unavailable until all the required indexes are explicitly created unless the CREATE TABLE statement is processed by the schema processor or the table space that contains the table is implicitly created. In that case, DB2 implicitly creates the indexes that are required for the unique keys and the table definition is considered complete. (For more information about implicitly created indexes, see Implicitly created indexes.)

references-clause

The *references-clause* of a *column-definition* provides a shorthand method of defining a foreign key composed of a single column. Thus, if *references-clause* is specified in the definition of column C, the effect is the same as if the *references-clause* were specified as part of a FOREIGN KEY clause in which C is the only identified column.

Do not specify *references-clause* in the definition of the following types of columns because these types of columns cannot be a foreign key:

- a LOB column
- a ROWID column
- a DECFLOAT column
- a distinct type column that is based on a LOB, ROWID, or DECFLOAT data type
- an XML column
- a row change timestamp column
- a security label column
- a Unicode column in an EBCDIC table

CHECK *(check-condition)*

CHECK *(check-condition)* provides a shorthand method of defining a check constraint that applies to a single column. For conformance with the SQL standard, if CHECK is specified in the column definition of column C, no columns other than C should be referenced in the check condition of the check constraint. The effect is the same as if the check condition were specified as a separate clause.

DEFAULT

Specifies the default value that is assigned to the column in the absence of a value specified on an insert or update operation or LOAD. DEFAULT must not be specified more than one time in the same *column-definition*. Do not specify DEFAULT for the following types of columns because DB2 generates default values:

- An identity column (a column that is defined AS IDENTITY)
- A ROWID column (or a distinct type that is based on a ROWID)
- A row change timestamp column

- A row-begin column
- A row-end column
- A *transaction-start-id* column
- An XML column

Do not specify a value after the DEFAULT keyword for a security label column. DB2 provides the default value for a security label column.

If a CCSID clause is specified for the column, do not specify a value after the DEFAULT keyword. Alternatively, DEFAULT NULL can be specified.

If a value is not specified after DEFAULT, the default value depends on the data type of the column, as follows:

Data Type
Default Value

Numeric
0

Big integer
0

Fixed-length character string
Blanks

Fixed-length graphic string
Blanks

Fixed-length binary string
Hexadecimal zeros

Varying-length string
A string of length 0

Inline BLOB
Hexadecimal zeros

Inline CLOB
Blanks

Inline DBCLOB
Blanks

Date CURRENT DATE

Time CURRENT TIME

TIMESTAMP(*integer*) WITHOUT TIME ZONE
CURRENT TIMESTAMP(*p*) WITHOUT TIME ZONE where *p* is the corresponding timestamp precision.

TIMESTAMP(*integer*) WITH TIME ZONE
CURRENT TIMESTAMP(*p*) WITH TIME ZONE where *p* is the corresponding timestamp precision.

If the column is defined as timestamp with time zone the default value must include a time zone.

Distinct type
The default of the source data type

A default value other than the one that is listed above can be specified in one of the following forms:
- WITH DEFAULT for a default value of an empty string
- DEFAULT NULL for a default value of null

Omission of NOT NULL and DEFAULT from a *column-definition*, for a column other than an identity column, is an implicit specification of DEFAULT NULL. For an identity column, it is an implicit specification of NOT NULL, and DB2 generates default values.

constant
Specifies a constant as the default value for the column. The value of the constant must conform to the rules for assigning that value to the column.

A character or graphic string constant must be short enough so that its UTF-8 representation requires no more than 1536. A hexadecimal graphic string constant (GX) cannot be specified.

In addition, the length of the constant value cannot be greater than the INLINE LENGTH attribute for LOB columns.

SESSION_USER or USER
Specifies the value of the SESSION_USER (USER) special register at the time of an SQL data change statement or LOAD as the default value for the column. If SESSION_USER is specified, the data type of the column must be a character string with a length attribute greater than or equal to 8 characters when the value is expressed in CCSID 37.If the data type of the column is an inline CLOB, the INLINE LENGTH attribute must be greater than or equal to 8 characters when the value is expressed as CCSID 37.

CURRENT SQLID
Specifies the value of the SQL authorization ID of the process at the time of an insert or update operation or LOAD as the default value for the column. If CURRENT SQLID is specified, the data type of the column must be a character string with a length attribute greater than or equal to the length attribute of the CURRENT SQLID special register.If the data type of the column is an inline CLOB, the INLINE LENGTH attribute must be greater than or equal to the length attribute of the CURRENT SQLID special register.

NULL
Specifies null as the default value for the column. If NOT NULL is specified, DEFAULT NULL must not be specified with the same *column-definition*.

cast-function-name
The name of the cast function that matches the name of the distinct type for the column. A cast function can only be specified if the data type of the column is a distinct type.

The schema name of the cast function, whether it is explicitly specified or implicitly resolved through function resolution, must be the same as the explicitly or implicitly specified schema name of the distinct type.

constant
Specifies a constant as the argument. The constant must conform to the rules of a constant for the source type of the distinct type.The length of the constant cannot be greater than the INLINE LENGTH attribute for LOB columns.

SESSION_USER or USER
Specifies the value of the SESSION_USER (USER) special register at the time a row is inserted as the default for the column. The source type of the distinct type of the column must be a CHAR, VARCHAR, or inline CLOB with a length attribute (inline length attribute for CLOB) that is greater than or equal to the length attribute of the SESSION_USER special register.

CURRENT SQLID
Specifies the value of the CURRENT SQLID special register at the time a row is inserted as the default for the column. The source type of the

distinct type of the column must be a CHAR, VARCHAR, or inline CLOB with a length attribute (or inline length attribute for CLOB) that is greater than or equal to the length attribute of the CURRENT SQLID special register.

NULL
Specifies the NULL value as the argument.

In a given column definition:

- DEFAULT and FIELDPROC cannot both be specified.
- NOT NULL and DEFAULT NULL cannot both be specified.

Table 117 summarizes the effect of specifying the various combinations of the NOT NULL and DEFAULT clauses on the CREATE TABLE statement *column-description* clause.

Table 117. Effect of specifying combinations of the NOT NULL and DEFAULT clauses

If NOT NULL is:	And DEFAULT is:	The effect is:
Specified[1]	Omitted	An error occurs if a value is not provided for the column on an insert or update operation or LOAD.
	Specified without an operand	The system defined nonnull default value is used.
	constant	The specified constant is used as the default value.
	SESSION_USER	The value of the SESSION_USER special register at the time of an insert or update operation or LOAD is used as the default value.
	CURRENT SQLID	The SQL authorization ID of the process at the time of an insert or update operation or LOAD is used as the default value.
	NULL	An error occurs during the execution of CREATE TABLE.
Omitted	Omitted	Equivalent to an implicit specification of DEFAULT NULL.
	Specified without an operand	The system defined nonnull default value is used.
	constant	The specified constant is used as the default value.
	SESSION_USER	The value of the SESSION_USER special register at execution time is used as the default value.
	CURRENT SQLID	The SQL authorization ID of the process is used as the default value.
	NULL	Null is used as the default value.

Note: The table does not apply to a column with a ROWID data type or to an identity column.

GENERATED

Specifies that DB2 generates values for the column. GENERATED must be specified if the column is to be considered one of the following types of columns:

- An identity column
- A row change timestamp column.
- A ROWID column
- A row-begin column
- A row-end column
- A *transaction-start-id* column

GENERATED must only be specified for these types of columns. GENERATED must not be specified with *default-clause* in a column definition.

GENERATED must not be specified if the column definition references global variables.

ALWAYS

Specifies that DB2 will always generate a value for the column when a row is inserted or updated and a default value must be generated. ALWAYS is the default and recommended value.

BY DEFAULT

Specifies that DB2 will generate a value for the column when a row is inserted or updated and a default value must be generated, unless an explicit value is specified.

For a row change timestamp column, DB2 inserts or updates a specified value but does not verify that the value is unique for the column unless the row change timestamp column has a unique constraint or a unique index that specifies only the row change timestamp column.

For a ROWID column, DB2 uses a specified value only if it is a valid row ID value that was previously generated by DB2 and the column has a unique, single-column index. Until this index is created on the ROWID column, the SQL insert or update operation and the LOAD utility cannot be used to add rows to the table. If the table space is explicitly created and the value of the CURRENT RULES special register is 'STD' when the CREATE TABLE statement is processed, or if the table space is implicitly created, DB2 implicitly creates the index on the ROWID column. The name of this index is 'I' followed by the first ten characters of the column name followed by seven randomly generated characters. If the column name is less than ten characters, DB2 adds underscore characters to the end of the name until it has ten characters. The implicitly created index has the COPY NO attribute.

For an identity column, DB2 inserts a specified value but does not verify that it a unique value for the column unless the identity column has a unique, single-column index.

BY DEFAULT is the recommended value only when you are using data propagation.

FOR EACH ROW ON UPDATE AS ROW CHANGE TIMESTAMP

Specifies that the column is a timestamp column for the table. DB2 generates a value for the column for each row as the row is inserted, and for any row in which any column is updated. The value that is generated for a row change timestamp column is a timestamp that corresponds to the

insert or update time of the row. If multiple rows are inserted or updated with a single statement, the value for the row change timestamp column might be different for each row.

A table can only have one row change timestamp column.

If *data-type* is specified, it must be TIMESTAMP WITHOUT TIME ZONE with a precision of 6.

A row change timestamp column cannot have a DEFAULT clause. NOT NULL must be specified for a row change timestamp column.

AS ROW BEGIN
Specifies that the column contains timestamp data and that the values are generated by DB2. DB2 generates a value for the column for each row as the row is inserted, and for every row in which any column is updated. The generated value is a timestamp that corresponds to the start time that is associated with the most recent transaction. If multiple rows are inserted with a single SQL statement, the values for the transaction start timestamp column are the same.

For a system-period temporal table, DB2 ensures the uniqueness of the generated values for a row-begin column across transactions. If multiple rows are inserted or updated within a single SQL transaction, the values for the row-begin column are the same for all the rows and are unique from the values that are generated for the column for another transaction. A row-begin column is required as the begin column of a SYSTEM_TIME period.

A table can have only one column defined as AS ROW BEGIN. If a data type is specified, it must be TIMESTAMP(12) WITHOUT TIME ZONE or TIMESTAMP(12) WITH TIME ZONE. If the column is defined as TIMESTAMP(12) WITH TIME ZONE, the values are stored in UTC, with a time zone of +00:00. If no data type is specified, the column is defined as TIMESTAMP(12) WITHOUT TIME ZONE. A column defined as a row-begin column cannot have a DEFAULT clause, and must be defined as NOT NULL. A row-begin column is not updatable.

A value for a row-begin column is composed of a TIMESTAMP(9) value that is unique per transaction per data sharing member followed by 3 digits that indicate the data sharing member number.

AS ROW END
Specifies that a value for the data type of the column is assigned by DB2 whenever a row is inserted or any column in the row is updated. The value that is assigned for a TIMESTAMP WITHOUT TIME ZONE column is the TIMESTAMP value '9999-12-30-00.00.00.000000000000'. The value that is assigned for a TIMESTAMP WITH TIME ZONE column is the TIMESTAMP value '9999-12-30-00.00.00.000000000000 +00:00'.

A row-end column is required as the second column of a SYSTEM_TIME period.

A table can have only one row-end column. If a data type is not specified, the column is defined as TIMESTAMP(12) WITHOUT TIME ZONE. If a data type is specified, it must be TIMESTAMP(12) WITHOUT TIME ZONE or TIMESTAMP(12) WITH TIME ZONE. If the column is defined as TIMESTAMP WITH TIME ZONE, the values are stored in UTC, with a time zone of +00:00. A row-end column cannot have a DEFAULT clause and must be defined as NOT NULL. A row-end column is not updatable.

AS TRANSACTION START ID

Specifies that the value is assigned by DB2 whenever a row is inserted into the table or any column in the row is updated. DB2 assigns a unique timestamp value per transaction or the null value. The null value is assigned to the transaction-start-ID column if the column is nullable. Otherwise, the value is generated using the time-of-day clock during execution of the first data change statement in the transaction that requires a value to be assigned to a row-begin column or transaction-start-ID column in the table, or when a row in a system-period temporal table is deleted. If multiple rows are inserted or updated within a single SQL transaction, the values for the transaction-start-ID column are the same for all the rows and are unique from the values that are generated for the column for another transaction.

A transaction-start-ID column is required for a system-period temporal table.

A table can have only one transaction-start-ID column. If a data type is not specified, the column is defined as TIMESTAMP(12) WITHOUT TIME ZONE. If a data type is specified, it must be TIMESTAMP(12) WITHOUT TIME ZONE or TIMESTAMP(12) WITH TIME ZONE. If the column is defined as TIMESTAMP WITH TIME ZONE, the values are stored in UTC, with a time zone of +00:00. A transaction-start-ID column cannot have a DEFAULT clause. A transaction-start-ID column is not updatable.

A value for a transaction-start-ID column is composed of a TIMESTAMP(9) value that is unique per transaction per data sharing member followed by 3 digits that indicate the data sharing member number.

PERIOD

Defines a period for the table. *begin-column-name* must not be the same as *end-column-name*. The data type, length, precision, and scale for *begin-column-name* must be the same as for *end-column-name*.

SYSTEM_TIME (*begin-column-name*,*end-column-name*)

Defines a system period with the name SYSTEM_TIME. There must not be a column in the table with the name SYSTEM_TIME. A table can have only one SYSTEM_TIME period. *begin-column-name* must be defined as AS ROW BEGIN and end-column-name must be defined as AS ROW END.

BUSINESS_TIME (*begin-column-name*,*end-column-name*)

Defines an application period with the name BUSINESS_TIME. There must not be a column in the table with the name BUSINESS_TIME. A table can have only one BUSINESS_TIME period. *begin-column-name* and *end-column-name* must be defined as DATE or TIMESTAMP(6) WITHOUT TIME ZONE, and the columns must be defined as NOT NULL. *begin-column-name* and *end-column-name* must not identify a column that is defined with a GENERATED clause.

A system generated check constraint named DB2_GENERATED_CHECK_CONSTRAINT_FOR_BUSINESS_TIME is generated to ensure that the value for *end-column-name* is greater than the value for *begin-column-name*. The table is placed in a check pending status.
DB2_GENERATED_CHECK_CONSTRAINT_FOR_BUSINESS_TIME must not be the name of an existing check constraint.

begin-column-name

Identifies the column that records the beginning of the period of time

in which a row is valid. The name must identify a column that exists in the table and must not be the same as a column that is used in the definition of another period for the table. *begin-column-name* must not be the same as *end-column-name*. The data type and precision for *begin-column-name* must be the same as for *end-column-name*.

For a SYSTEM_TIME period, *begin-column-name* must be defined as AS ROW BEGIN.

For a BUSINESS_TIME period, the column must not be defined with a GENERATED clause.

end-column-name
Identifies the column that records the end of the period of time in which a row is valid. In the history table that is associated with a system-period temporal table, the history table column that corresponds to *end-column-name* in the system-period temporal table is set to reflect the deletion of the row. The name must identify a column that exists in the table and must not be the same as a column that is used in the definition of another period for the table.

For a SYSTEM_TIME period, *end-column-name* must be defined as AS ROW END.

For a BUSINESS_TIME period, the column must not be defined with a GENERATED clause.

AS IDENTITY
Specifies that the column is an identity column for the table. A table can have only one identity column. AS IDENTITY can be specified only if the data type for the column is an exact numeric type with a scale of zero (SMALLINT, INTEGER, BIGINT, DECIMAL with a scale of zero, or a distinct type based on one of these types).

An identity column is implicitly NOT NULL. An identity column cannot have a WITH DEFAULT clause.

Defining a column AS IDENTITY does not necessarily ensure the uniqueness of the values. To ensure uniqueness of the values, define a unique, single-column index on the identity column.

START WITH *numeric-constant*
Specifies the first value that is generated for the identity column. The value can be any positive or negative value that could be assigned to the column without non-zero digits existing to the right of the decimal point.

If a value is not explicitly specified when the identity column is defined, the default is the MINVALUE for an ascending identity column and the MAXVALUE for a descending identity column. This value is not necessarily the value that would be cycled to after reaching the maximum or minimum value for the identity column. The START WITH clause can be used to start the generation of values outside the range that is used for cycles. The range used for cycles is defined by MINVALUE and MAXVALUE.

INCREMENT BY *numeric-constant*
Specifies the interval between consecutive values of the identity column. The value can be any positive or negative value (including 0)

that does not exceed the value of a large integer constant, and could be assigned to the column without any non-zero digits existing to the right of the decimal point.

If this value is negative, the values for the identity column descend. If this value is 0 or positive, the values for the identity column ascend. The default is 1.

MINVALUE or NO MINVALUE
Specifies the minimum value at which a descending identity column either cycles or stops generating values or an ascending identity column cycles to after reaching the maximum value.

NO MINVALUE
Specifies that the minimum end point of the range of values for the identity column has not be set. In such a case, the default value for MINVALUE becomes one of the following:

- For an ascending identity column, the value is the START WITH value or 1 if START WITH is not specified.
- For a descending identity column, the value is the minimum value of the data type of the column.

The default is NO MINVALUE.

MINVALUE *numeric-constant*
Specifies the numeric constant that is the minimum value that is generated for this identity column. This value can be any positive or negative value that could be assigned to this column without non-zero digits existing to the right of the decimal point. The value must be less than or equal to the maximum value.

MAXVALUE or NO MAXVALUE
Specifies the maximum value at which an ascending identity column either cycles or stops generating values or a descending identity column cycles to after reaching the minimum value.

NO MAXVALUE
Specifies that the minimum end point of the range of values for the identity column has not be set. In such a case, the default value for MAXVALUE becomes one of the following:

- For an ascending identity column, the value is the maximum value of the data type associated with the column.
- For a descending identity column, the value is the START WITH value -1 if START WITH is not specified.

The default is NO MAXVALUE.

MAXVALUE *numeric-constant*
Specifies the numeric constant that is the maximum value that is generated for this identity column. This value can be any positive or negative value that could be assigned to this column without non-zero digits existing to the right of the decimal point. The value must be greater than or equal to the minimum value.

CYCLE or NO CYCLE
Specifies whether this identity column should continue to generate values after reaching either its maximum or minimum value. The default is NO CYCLE.

NO CYCLE
Specifies that values will not be generated for the identity column after the maximum or minimum value has been reached.

CYCLE
Specifies that values continue to be generated for the identity column after the maximum or minimum value has been reached. If this option is used, after an ascending identity column reaches the maximum value, it generates its minimum value. After a descending identity column reaches its minimum value, it generates its maximum value. The maximum and minimum values for the identity column determine the range that is used for cycling.

When CYCLE is in effect, duplicate values can be generated by DB2 for an identity column. However, if a unique index exists on the identity column and a non-unique value is generated for it, an error occurs.

CACHE *integer-constant* **or NO CACHE**
Specifies whether to keep some preallocated values in memory. Preallocating and storing values in the cache improves the performance of inserting rows into a table. The default is **CACHE 20**.

NO CACHE
Specifies that values for the identity column and sequences are not preallocated and stored in the cache, ensuring that values will not be lost in the case of a system failure. In this case, every request for a new value for the identity column or sequence results in synchronous I/O.

In a data sharing environment, use **NO CACHE** if you need to guarantee that the identity column and sequence values are generated in the order in which they are requested.

CACHE *integer-constant*
Specifies the maximum number of values of the identity column sequence that DB2 can preallocate and keep in memory.

During a DB2 shutdown, all cached identity column values and sequence values that are yet to be assigned will be lost and will not be used. Therefore, the value that is specified for **CACHE** also represents the maximum number of identity column values and sequence values that will be lost during a DB2 shutdown.

The minimum value is 2.

In a data sharing environment, you can use the **CACHE** and **NO ORDER** options to allow multiple DB2 members to cache sequence values simultaneously.

ORDER or NO ORDER
Specifies whether the identity column values must be generated in order of request. The default is NO ORDER.

NO ORDER
Specifies that the values do not need to be generated in order of request.

ORDER
Specifies that the values are generated in order of request.

Specifying ORDER might disable the caching of values. ORDER applies only to a single-application process.

In a data sharing environment, if the CACHE and NO ORDER options are in effect, multiple caches can be active simultaneously, and the requests for identity values from different DB2 members might not result in the assignment of values in strict numeric order. For example, if members DB2A and DB2B are using the identity column, and DB2A gets the cache values 1 to 20 and DB2B gets the cache values 21 to 40, the actual order of values assigned would be 1,21,2 if DB2A requested a value first, then DB2B requested, and then DB2A again requested. Therefore, to guarantee that identity values are generated in strict numeric order among multiple DB2 members using the same identity column, specify the ORDER option.

FIELDPROC *program-name*

Designates *program-name* as the field procedure exit routine for the column. A field procedure can be specified only for a column with a length attribute that is not greater than 255 bytes. FIELDPROC can only be specified for columns that are a built-in character string or graphic string data types. The column must not be one of the following:

- a LOB column
- a security label column
- a row change timestamp column
- a column with the TIMESTAMP WITH TIME ZONE data type
- a Unicode column in an EBCDIC table

The field procedure encodes and decodes column values: before a value is inserted in the column, it is passed to the field procedure for encoding. Before a value from the column is used by a program, it is passed to the field procedure for decoding. A field procedure could be used, for example, to alter the sorting sequence of values entered in the column.

The field procedure is also invoked during the processing of the CREATE TABLE statement. When so invoked, the procedure provides DB2 with the column's *field description*. The field description defines the data characteristics of the encoded values. By contrast, the information you supply for the column in the CREATE TABLE statement defines the data characteristics of the decoded values.

Related information:

Field procedures (DB2 Administration Guide)

"Character and graphic string comparisons" on page 135

constant

Is a parameter that is passed to the field procedure when it is invoked. A parameter list is optional. The *n*th parameter specified in the FIELDPROC clause on CREATE TABLE corresponds to the *n*th parameter of the specified field procedure. The maximum length of the parameter list is 254 bytes, including commas but excluding insignificant blanks and the delimiting parentheses.

If you omit FIELDPROC, the column has no field procedure.

AS SECURITY LABEL

Specifies that the column will contain security label values. This also indicates that the table is defined with multilevel security with row level granularity. A

table can have only one security label column. To define a table with a security label column, the primary authorization ID of the statement must have a valid security label, and the RACF SECLABEL class must be active. In addition, the following conditions are also required:

- The data type of the column must be CHAR(8).
- The subtype of the column must be SBCS.
- The column must be defined with the NOT NULL and WITH DEFAULT clauses.
- The WITH DEFAULT clause must not specify a default value (DB2 determines the default value)
- No field procedures, check constraints, or referential constraints are defined on the column.
- No edit procedure for the table can be defined with row attribute sensitivity.

For information about using multilevel security, see *DB2 Administration Guide*.

IMPLICITLY HIDDEN
Specifies that the column is not visible in the result for SQL statements unless you explicitly refer to the column by name. For example, assuming that the table T1 includes a column that is defined with the IMPLICITLY HIDDEN clause, the result of a SELECT * would not include the implicitly hidden column. However, the result of a SELECT statement that explicitly refers to the name of the implicitly hidden column would include that column in the result table.

IMPLICITLY HIDDEN must not be specified for a column that is defined as a ROWID, or a distinct type that is based on a ROWID. IMPLICITLY HIDDEN must not be specified for all columns of a table.

INLINE LENGTH *integer*
Specifies the maximum length for the column, if the column is a LOB column and the table is in a universal table space. INLINE LENGTH cannot be specified if the column is not a LOB column (or a distinct type that is based on a LOB) or if the table is not in a universal table space.

For BLOB and CLOB columns, *integer* specifies the maximum number of bytes that are stored in the base table space for the column. *integer* must be between 0 and 32680 (inclusive) for a BLOB or CLOB column.

For a DBCLOB column, *integer* specifies the maximum number of double-byte characters that are stored in the table space for the column. *integer* must be between 0 and 16340 (inclusive) for a DBCLOB column.

If INLINE LENGTH is specified, the value of *integer* cannot be greater than the maximum length of the LOB column.

If the INLINE LENGTH clause is not specified, the maximum length of the LOB column depends on the following conditions:

- If a distinct type is not used or the distinct type that is used has been created without the INLINE LENGTH attribute, the LOB column will use the value of the LOB INLINE LENGTH parameter on installation panel DSNTIPD as the default inline length when the value of LOB INLINE LENGTH does not exceed the maximum length of the LOB column. If the value of LOB INLINE LENGTH exceeds the maximum length of the LOB column, the maximum length is the inline length of this LOB column.
- If a distinct type that has been created with the INLINE LENGTH attribute is used, the LOB column inherits the inline length from the distinct type.

Regardless of how the length is determined, the inline length of the LOB cannot be greater than its maximum length.

CONSTRAINT *constraint-name*
Names the constraint. If a constraint name is not specified, a unique constraint name is generated. If a name is specified, it must be different from the names of any referential, check, primary key, or unique key constraints previously specified on the table.

PRIMARY KEY(*column-name*,...)
Defines a primary key composed of the identified columns. The clause must not be specified more than one time and the same column must not be identified more than one time. The identified columns must be defined as NOT NULL. Each *column-name* must be an unqualified name that identifies a column of the table except for the following types of columns:

- a LOB column
- a ROWID column
- a DECFLOAT column
- a distinct type column that is based on a LOB, ROWID, or DECFLOAT
- an XML column
- a row change timestamp column
- a Unicode column in an EBCDIC table

The number of identified columns must not exceed 64. In addition, the sum of the length attributes of the columns must not be greater than 2000 - 2*m*, where *m* is the number of varying-length columns in the key.

The table is marked as unavailable until its primary index is explicitly created unless the table space is explicitly created and the CREATE TABLE statement is processed by the schema processor, or the table space is implicitly created. In that case, DB2 implicitly creates an index to enforce the uniqueness of the primary key and the table definition is considered complete. (For more information about implicitly created indexes, see Implicitly created indexes.)

BUSINESS_TIME WITHOUT OVERLAPS can be specified as the last item in the list. If BUSINESS_TIME WITHOUT OVERLAPS is specified, the list must include at least one *column-name* or *key-expression*. When WITHOUT OVERLAPS is specified, the values for the rest of the specified keys are unique with respect to the time for the BUSINESS_TIME period. When BUSINESS_TIME WITHOUT OVERLAPS is specified, the columns of the BUSINESS_TIME period must not be specified as part of the constraint. The specification of BUSINESS_TIME WITHOUT OVERLAPS adds the following to the constraint:

- The end column of the BUSINESS_TIME period in ascending order
- The begin column of the BUSINESS_TIME period in ascending order

UNIQUE(*column-name*,...**)**
Defines a unique key composed of the identified columns. Each *column-name* must be an unqualified name that identifies a column of the table. Each identified column must be defined as NOT NULL. The same column must not be identified more than one time. The following types of columns cannot be identified:

- a LOB column
- a ROWID column
- a DECFLOAT column
- a distinct type column that is based on a LOB, ROWID, or DECFLOAT
- a row change timestamp column
- a Unicode column in an EBCDIC table

The number of identified columns must not exceed 64. In addition, the sum of the length attributes of the columns must not be greater than 2000 - 2*m*, where *m* is the number of varying-length columns in the key.

A unique key is a duplicate if it is the same as the primary key or a previously defined unique key. The specification of a duplicate unique key is ignored with a warning.

The table is marked as unavailable until all the required indexes are explicitly created unless the table space is explicitly created and the CREATE TABLE statement is processed by the schema processor, or the table space is implicitly created. In these cases, DB2 implicitly creates the indexes that are required for the unique keys and the table definition is considered complete. (For more information about implicitly created indexes, see Implicitly created indexes.)

BUSINESS_TIME WITHOUT OVERLAPS can be specified as the last item in the list. If BUSINESS_TIME WITHOUT OVERLAPS is specified, the list must include at least one *column-name* or *key-expression*. When WITHOUT OVERLAPS is specified, the values for the rest of the specified keys are unique with respect to the time for the BUSINESS_TIME period. When BUSINESS_TIME WITHOUT OVERLAPS is specified, the columns of the BUSINESS_TIME period must not be specified as part of the constraint. The specification of BUSINESS_TIME WITHOUT OVERLAPS adds the following to the constraint:

- The end column of the BUSINESS_TIME period in ascending order
- The begin column of the BUSINESS_TIME period in ascending order

CONSTRAINT *constraint-name*
Names the referential constraint. If a constraint name is not specified, a unique constraint name is generated. If a name is specified, it must be different from the names of any referential, check, primary key, or unique key constraints previously specified on the table.

FOREIGN KEY (*column-name***,...) references-clause**
Each specification of the FOREIGN KEY clause defines a referential constraint.

The foreign key of the referential constraint is composed of the identified columns. Each *column-name* must be an unqualified name that identifies a column of the table. The same column must not be identified more than one time. The column cannot be any of the followinf types of columns:

- a LOB column
- a ROWID column
- a DECFLOAT column
- an XML column
- a row change timestamp column
- a security label column
- a Unicode column in an EBCDIC table

The number of identified columns must not exceed 64. The sum of the column length attributes must not exceed 255 minus the number of columns that allow null values. The referential constraint is a duplicate if the FOREIGN KEY and parent table are the same as the FOREIGN KEY and parent table of a previously defined referential constraint. The specification of a duplicate referential constraint is ignored with a warning.

REFERENCES *table-name (column-name,...)*

The table name that is specified after REFERENCES must identify a table that exists at the current server[32]. The table name must not identify one of the following tables:

- A catalog table
- A declared global temporary table
- A history table
- An archive table

In the following discussion, let T2 denote an identified table and let T1 denote the table that you are creating (T1 and T2 cannot be the same table[32]).

T2 must have a unique index. The privilege set must include the ALTER or REFERENCES privilege on the parent table, or the REFERENCES privilege on the columns of the nominated parent key.

The parent key of the referential constraint is composed of the identified columns. Each *column-name* must be an unqualified name that identifies a column of T2. The same column must not be identified more than one time. The identified column cannot be any of the following types of columns:

- a LOB column
- a ROWID column
- a DECFLOAT column
- an XML column
- a row change timestamp column
- a security label column
- a Unicode column in an EBCDIC table

The list of column names in the parent key must match the list of column names in a primary key or unique key in the parent table T2. The column names must be specified in the same order as in the primary key or unique key. If any of the referenced columns in T2 has a non-numeric data type, T2 and T1 must use the same encoding scheme.

If a list of column names is not specified, T2 must have a primary key. Omission of a list of column names is an implicit specification of the columns of the primary key for T2.

The specified foreign key must have the same number of columns as the parent key of T2. The description of the *n*th column of the foreign key must be identical to the description of the *n*th column of the nominated parent key. The exception is that their names, default values, null attributes, and check constraints do not have to match. If the foreign key includes a column that is defined as a distinct type, the corresponding column of the nominated parent key must be the same distinct type. If a column of the foreign key has a field procedure, the corresponding column of the nominated parent key must have the same field procedure and an identical field description. A field description is a description of the encoded value as it is stored in the database for a column that is defined to have an associated field procedure.

The referential constraint that is specified by a FOREIGN KEY clause defines a relationship in which T2 is the parent and T1 is the dependent. A description of the referential constraint is recorded in the catalog.

32. This restriction is relaxed when the statement is processed by the schema processor and the other table is created within the same CREATE SCHEMA.

ON DELETE

The delete rule of the relationship is determined by the ON DELETE clause. For more on the concepts used here, see "Referential constraints" on page 23.

SET NULL must not be specified unless some column of the foreign key allows null values. The default value for the rule depends on the value of the CURRENT RULES special register when the CREATE TABLE statement is processed. If the value of the register is 'DB2', the delete rule defaults to RESTRICT; if the value is 'STD', the delete rule defaults to NO ACTION.

The delete rule applies when a row of T2 is the object of a DELETE or propagated delete operation and that row has dependents in T1. Let *p* denote such a row of T2. Then:

- If RESTRICT or NO ACTION is specified, an error occurs and no rows are deleted.
- If CASCADE is specified, the delete operation is propagated to the dependents of *p* in T1.
- If SET NULL is specified, each nullable column of the foreign key of each dependent of *p* in T1 is set to null.

Let T3 denote a table identified in another FOREIGN KEY clause (if any) of the CREATE TABLE statement. The delete rules of the relationships involving T2 and T3 must be the same and must not be SET NULL if:

- T2 and T3 are the same table.
- T2 is a descendent of T3 and the deletion of rows from T3 cascades to T2.
- T2 and T3 are both descendents of the same table and the deletion of rows from that table cascades to both T2 and T3.

ENFORCED or NOT ENFORCED

Indicates whether or not the referential constraint is enforced by DB2 during normal operations, such as insert, update, or delete.

ENFORCED

Specifies that the referential constraint is enforced by the DB2 during normal operations (such as insert, update, or delete) and that it is guaranteed to be correct. This is the default.

NOT ENFORCED

Specifies that the referential constraint is not enforced by DB2 during normal operations, such as insert, update, or delete. This option should only be used when the data that is stored in the table is verified to conform to the constraint by some other method than relying on the database manager.

ENABLE QUERY OPTIMIZATION

Specifies that the constraint can be used for query optimization. DB2 uses the information in query optimization using materialized query tables with the assumption that the constraint is correct. This is the default.

check-constraint

CONSTRAINT *constraint-name*

Names the check constraint. The constraint name must be different from the names of any referential, check, primary key, or unique key constraints previously specified on the table.

If *constraint-name* is not specified, a unique constraint name is derived from the name of the first column in the *check-condition* specified in the definition of the check constraint.

CHECK (*check-condition***)**

Defines a check constraint. At any time, the *check-condition* must be true or unknown for every row of the table. A *check-condition* can evaluate to unknown if a column that is an operand of the predicate is null. A *check-condition* that evaluates to unknown does not violate the check constraint. A *check-condition* is a search condition, with the following restrictions:

- It can refer only to columns of table *table-name*.
- The columns cannot be the following types of columns:
  - LOB columns
  - ROWID columns
  - DECFLOAT columns
  - distinct type columns that are based on LOB, ROWID, and DECFLOAT data types
  - XML columns
  - security label columns
  - Unicode columns in an EBCDIC table
- It can be up to 3800 bytes long, not including redundant blanks.
- It must not contain any of the following:
  - Subselects
  - Built-in or user-defined functions
  - CAST specifications
  - Cast functions other than those created when the distinct type was created
  - Host variables
  - Parameter markers
  - Special registers
  - Global variables
  - Columns that include a field procedure
  - CASE expressions
  - ROW CHANGE expressions
  - Row expressions
  - DISTINCT predicates
  - GX constants (hexadecimal graphic string constants)
  - Sequence references
  - OLAP specifications
- If a *check-condition* refers to a LOB column (including a distinct type that is based on a LOB), the reference must occur within a LIKE predicate.
- The AND and OR logical operators can be used between predicates. The NOT logical operator cannot be used.
- The first operand of every predicate must be the column name of a column in the table.
- The second operand in the *check-condition* must be either a constant or a column name of a column in the table.
  - If the second operand of a predicate is a constant, and if the constant is:
    - A floating-point number, then the column data type must be floating point.

- A decimal number, then the column data type must be either floating point or decimal.
- An integer number, then the column data type must not be a small integer.
- A small integer number, then the column data type must be small integer.
- A decimal constant, then its precision must not be larger than the precision of the column.

– If the second operand of a predicate is a column, then both columns of the predicate must have:
  - The same data type.
  - Identical descriptions with the exception that the specification of the NOT NULL and DEFAULT clauses for the columns can be different, and that string columns with the same data type can have different length attributes.

ORGANIZE BY HASH

Specifies that a hash is to be used for the data organization of the table.

If PARTITION BY RANGE is specified, and the IN clause specifies a table space, the table space must be a partition by range universal table space.

If PARTITION BY RANGE is not specified, and an IN clause is specified, the IN clause must identify a partition-by-growth universal table space.

ORGANIZE BY HASH must not be specified if the table is defined with APPEND YES.

ORGANIZE BY HASH must not be specified if the table is using basic row format.

UNIQUE

Specifies that DB2 enforces uniqueness of the hash key columns, preventing the table from containing two or more rows with the same value of the hash key.

(*column-name*,...)

The list of column names defines the hash key that is used to determine where a row will be placed. Each *column-name* must be an unqualified name that identifies a column of the table. The same column must not be specified more than once and the specified columns must be defined as NOT NULL. The number of specified columns must not exceed 64, and the sum of their length attributes must not exceed 255. A specified column cannot be any of the following types:

- a LOB column
- a DECFLOAT column
- a XML column
- a distinct type column that is based on one of the preceding data types
- a Unicode column in an EBCDIC table

If PARTITION BY RANGE is also specified, the list of column names must specify all of the column names that are specified in *partition-expression* for the table, and must specify the column names in the same order as *partition-expression*. If the ORGANIZE BY clause contains more columns than *partition-expression*, *partition-expression* determines the partition number.

HASH SPACE *integer***K|M|G**

Specifies the amount of fixed hash space to preallocate for the table. If the table is partitioned by range, this is the space for each partition.

The default is 64M for a table in a partition-by-growth universal table space or 64M for each partition of a partition by range universal table space.

K Indicates that the integer value is to be multiplied by 1024 to specify the hash space size in bytes. The integer must be between 256 and 268435456.

M Indicates that the integer value is to be multiplied by 1048576 to specify the hash space size in bytes. The integer must be between 1 and 262144.

G Indicates that the integer value is to be multiplied by 1073741824 to specify the hash space size in bytes. The integer must be between 1 and 256 for a partition by range table and must be between 1 and 131072 for a non-partitioned table.

If a value greater than 4G is specified, the data sets for the table space are associated with a DFSMS data class that has been specified with extended format and extended addressability.

LIKE *table-name* **or** *view-name*

Specifies that the columns of the table have exactly the same name and description as the columns of the identified table or view.

The name that is specified after LIKE must identify a table or view that exists at the current server or a declared temporary table. A view cannot contain columns of length 0.

table-name or *view-name* must not contain a Unicode column in an EBCDIC table.

The privilege set must implicitly or explicitly include the SELECT privilege on the identified table or view. If the identified table or view contains a column with a distinct type, the USAGE privilege on the distinct type is also needed. An identified table must not be an auxiliary table or a clone table. An identified view must not include a column that is an explicitly defined ROWID column (including a distinct type that is based on a ROWID), an identity column, or a row change timestamp column.

The use of LIKE is an implicit definition of *n* columns, where *n* is the number of columns in the identified table (including implicitly hidden columns) or view. A column of the new table that corresponds to an implicitly hidden column in the existing table will also be defined as implicitly hidden. The implicit definition includes all attributes of the *n* columns as they are described in SYSCOLUMNS with the following exceptions:

- When a table is identified in the LIKE clause and a column in the table has a field procedure, the corresponding column of the new table has the same field procedure and the field description. However, the field procedure is not invoked during the execution of the CREATE TABLE statement. When a view is identified in the LIKE clause, none of the columns of the new table will have a field procedure. This is true even in the case that a column of a base table underlying the view has a field procedure defined.
- When a table is identified in the LIKE clause and a column in the table is an identity column, the corresponding column of the new table inherits only

the data type of the identity column; none of the identity attributes of the column are inherited unless the INCLUDING IDENTITY clause is specified.

- When a table is identified in the LIKE clause and a column in the table is a security label column, the corresponding column of the new table inherits only the data type of the security label column; none of the security label attributes of the column are inherited.
- When a table is identified in the LIKE clause and the table contains a ROWID column (explicitly-defined or implicitly hidden), the corresponding columns of the new table inherits the ROWID columns.
- When a table is identified in the LIKE clause and the table contains row change timestamp column, a transaction-start-ID column, a row-begin column, or a row-end column, the corresponding column of the new table inherits only the data type of the original column. The new column is not considered a generated column.
- When a table is identified in the LIKE clause and the table contains an inline LOB column, the corresponding columns of the new table will inherit the inline attribute if the table is in an universal table space. Otherwise, the inline attribute of the table identified in the LIKE clause is ignored.
- When a view is identified in the LIKE clause, the default value that is associated with the corresponding column of the new table depends on the column of the underlying base table for the view. If the column of the base table does not have a default, the new column does not have a default. If the column of the base table has a default, the default of the new column is:
  - Null if the column of the underlying base table allows nulls.
  - The default for the data type of the underlying base table if the underlying base table does not allow nulls.

  The above defaults are chosen regardless of the current default of the base table column. The existence of an INSTEAD OF trigger does not affect the inheritance of default values.
- When a table that uses table-controlled partitioning is identified in the LIKE clause, the new table does not inherit partitioning scheme of that table. You can add these partition boundaries by specifying ALTER TABLE with the ADD PARTITION BY RANGE clause.
- The CCSID of the column is determined by the implicit or explicit CCSID clause. For more information, see the CCSID clause.
- When a table is identified in the LIKE clause and the table includes a period, the new table does not inherit the period.
- When the table that is identified in the LIKE clause is a system-period temporal table, the new table is not a system-period temporal table.
- When the table that is identified in the LIKE clause has row access controls or column access controls activated, the new table does not inherit the row access controls or the column access controls.

The implicit definition does not include any other attributes of the identified table or view. For example, the new table does not have a primary key or foreign key. The table is created in the table space implicitly or explicitly specified by the IN clause, and the table has any other optional clause only if the optional clause is specified.

AS *(fullselect)*
Specifies that the table definition is based on the column definitions from the result of a query expression. The use of AS *(fullselect)* is an implicit definition of *n* columns for the table, where *n* is the number of columns that would result

from the *fullselect*. The columns of the new table are defined by the columns that result from the *fullselect*. Every select list element must have a unique name. The AS clause can be used in the *select-clause* to provide unique names.

The implicit definition includes the column name, data type, length, precision, scale, and nullability characteristic of each of the result columns of *fullselect*. The length of each column must not be 0. Other column attributes, such as DEFAULT and IDENTITY, are not inherited from the *fullselect*. A column of the new table that corresponds to an implicitly hidden column of a base table referenced in the *fullselect* is not considered hidden in the new table. A FIELDPROC is inherited for a column if the corresponding select item of the *fullselect* is a column that can be mapped to a column of a base table or a view. The new table contains a security label column if only one table in the *fullselect* contains a security label column and the primary authorization ID of the statement has a valid security label. If more than one table in the *fullselect* contains a security label column, an error occurs.

The implicit definition does not include any other attributes of the identified table or view. For example, the new table does not have a primary key or foreign key. The table is created in the table space implicitly or explicitly specified by the IN clause, and the table has any other optional clause only if the optional clause is specified.

The owner of the table being created must have the SELECT privilege on the tables or views referenced in the fullselect, or the privilege set must include SYSADM or DBADM authority for the database in which the tables of the fullselect reside. Having SELECT privilege means that the owner has at least one of the following authorizations:

- Ownership of the tables or views referenced in the fullselect
- The SELECT privilege on the tables and views referenced in the fullselect
- SYSADM authority
- DBADM authority for the database in which the tables of the fullselect reside

The rules for establishing the qualifiers for names used in the fullselect are the same as the rules used to establish the qualifiers for *table-name*.

The fullselect must not:

- Result in a column having a ROWID, BLOB, CLOB, DBCLOB, or XML data type or a distinct type based on these data types.
- Include multiple security label columns.
- Include a PREVIOUS VALUE or a NEXT VALUE expression.
- Refer to host variables or include parameter markers.
- Include an SQL data change statement in the FROM clause.
- Reference data that is encoded with different CCSID sets.
- Result in a column that is an array
- Reference a remote object.

If the WITH NO DATA clause is not specified, the new table is considered a materialized query table.

WITH NO DATA

Specifies that the query is used only to define the attributes of the new table. The table is not populated using the results of the query and the REFRESH TABLE statement cannot be used.

If the tables that are specified in the *fullselect* use row access controls or column access controls, the row access controls and the column access controls are not defined for the new table.

copy-options
Specifies whether identity column attributes, row change timestamp attributes, and column defaults are inherited from the definition of the source of the result table.

EXCLUDING IDENTITY COLUMN ATTRIBUTES or INCLUDING IDENTITY COLUMN ATTRIBUTES
Specifies whether identity column attributes are inherited from the definition of the source of the result table.

EXCLUDING IDENTITY COLUMN ATTRIBUTES
Specifies that identity column attributes are not inherited from the definition of the source of the result table. This is the default.

INCLUDING IDENTITY COLUMN ATTRIBUTES
Specifies that, if available, identity column attributes (such as START WITH, INCREMENT BY, and CACHE values) are inherited from the definition of the source table. These attributes can be inherited if the element of the corresponding column in the table, view, or *fullselect* is the name of a column of a table or the name of a column of a view that directly or indirectly maps to the column name of a base table with the identity attribute. In other cases, the columns of the new temporary table do not inherit the identity attributes. The columns of the new table do not inherit the identity attributes in the following cases:

- The select list of the *fullselect* includes multiple instances of an identity column name (that is, selecting the same column more than one time).
- The select list of the *fullselect* includes multiple identity columns (that is, it involves a join).
- The identity column is included in an expression in the select list.
- The *fullselect* includes a set operation.

EXCLUDING ROW CHANGE TIMESTAMP COLUMN ATTRIBUTES or INCLUDING ROW CHANGE TIMESTAMP COLUMN ATTRIBUTES
Specifies whether row change timestamp column attributes are inherited from the definition of the source of the result table.

EXCLUDING ROW CHANGE TIMESTAMP COLUMN ATTRIBUTES
Specifies that row change timestamp column attributes are not inherited from the source result table definition. This is the default.

INCLUDING ROW CHANGE TIMESTAMP COLUMN ATTRIBUTES
Specifies that, if available, row change timestamp column attributes are inherited from the definition of the source table. These attributes can be inherited if the element of the corresponding column in the table, view, or *fullselect* is the name of a column of a table or the name of a column of a view that directly or indirectly maps to the column name of a base table defined as a row change timestamp column. In other cases, the columns of the new temporary table do not inherit the row change timestamp column attributes. The columns of the new table do not inherit the row change timestamp attributes in the following cases:

- The select list of the *fullselect* includes multiple instances of a row change timestamp column name (that is, selecting the same column more than one time).
- The select list of the *fullselect* includes multiple row change timestamp column names (that is, it involves a join).
- The row change timestamp column is included in an expression in the select list.
- The *fullselect* includes a set operation (such as union).

EXCLUDING COLUMN DEFAULTS, INCLUDING COLUMN DEFAULTS, or USING TYPE DEFAULTS

Specifies whether column defaults are inherited from the source result table definition. EXCLUDING COLUMN DEFAULTS, INCLUDING COLUMN DEFAULTS, and USING TYPE DEFAULTS must not be specified if the LIKE clause is specified.

EXCLUDING COLUMN DEFAULTS

Specifies that the column defaults are not inherited from the definition of the source table. The default values of the column of the new table are either null or there are no default values. If the column can be null, the default is the null value. If the column cannot be null, there is no default value, and an error occurs if a value is not provided for a column on an insert or update operation, or LOAD for the new table.

INCLUDING COLUMN DEFAULTS

Specifies that column defaults for each updatable column of the definition of the source table are inherited. Columns that are not updatable do not have a default defined in the corresponding column of the created table. The existence of an INSTEAD OF trigger for a view does not affect the inheritance of default values.

USING TYPE DEFAULTS

Specifies that the default values for the table depend on data type of the columns that result from *fullselect*, as follows:

Data type
: **Default value**

Numeric
: 0

Fixed-length character string
: Blanks

Fixed-length graphic string
: Blanks

Fixed-length binary string
: Hexadecimal zeros

Varying-length string
: A string of length 0

Fixed-length char or fixed-length graphic
: A string of blanks

Fixed-length binary
: Hexadecimal zeros

Date CURRENT DATE

Time CURRENT TIME

Timestamp(*integer*) without time zone
: CURRENT TIMESTAMP(*p*) WITHOUT TIME ZONE where *p* is the corresponding timestamp precision.

Timestamp(*integer*) with time zone
CURRENT TIMESTAMP(*p*) WITH TIME ZONE where *p* is the corresponding timestamp precision.

WITH RESTRICT ON DROP
Indicates that the table can be dropped only by using REPAIR DBD DROP. In addition, the database and table space that contain the table can be dropped only by using REPAIR DBD DROP.

The WITH RESTRICT ON DROP clause can be removed using the ALTER TABLE statement with the DROP RESTRICT ON DROP clause. After the WITH RESTRICT ON DROP clause is removed from the definition of the table, the table, the database, and the containing table space can be dropped using the DROP statement.

IN *database-name.table-space-name* **or IN DATABASE** *database-name*
Identifies the database and table space in which the table is created. Both forms are optional.

If you specify both a database and a table space, the database must be described in the catalog on the current server. The database must not be DSNDB06 or a work file database. The table space must belong to the database that you specify and must not be an XML table space.

If you specify a database but not a table space, a table space is implicitly created in *database-name*. The name of the table space is derived from the name of the table. The qualifier of the table space is the same as the qualifier of the table. The buffer pool that is used is the default buffer pool for user data that is specified on installation panel DSNTIP1. If you specify a table space but not a database, the database that contains the table space is used.

If you specify neither a table space or a database, a database is implicitly created with the name DSN*xxxxx*, where *xxxxx* is a five-digit number. A table space is also implicitly created.

If you specify a table space, it cannot be one of the following table spaces:

- A table space that was created implicitly
- A partitioned or a partition-by-growth table space that already contains a table
- A LOB table space
- A table space that already contains a system-period temporal table, a history table, an archive-enabled table, or an archive table

If you specify a partitioned table space, you cannot load or use the table until its partitioned scheme is created.

You cannot specify the name of an implicitly created database, That is, you specify a database name that is eight characters, DSN*xxxxx*, where *xxxxx* is a five-digit number.

To create a table space implicitly, the privilege set must have: SYSADM or SYSCTRL authority; DBADM, DBCTRL, or DBMAINT authority for the database; or the CREATETS privilege for the database. You must also have the USE privilege for the default buffer pool in the database and default storage group.

If you specify a table space name, you must have SYSADM or SYSCTRL authority, DBADM authority for the database, or the USE privilege for the table space.

PARTITION BY RANGE or PARTITION BY SIZE
Specifies the partitioning scheme for the table.

PARTITION BY RANGE
Specifies the range partitioning scheme for the table (the columns that are used to partition the data). When this clause is specified, the table space is complete, and it is not necessary to create a partitioned index on the table. If this clause is used, the ENDING AT clause cannot be used on a subsequent CREATE INDEX statement for this table.

If this clause is specified, the IN *database-name.table-space-name* clause is required. This clause applies only to tables in a partitioned table space. PARTITION BY RANGE must not be specified for a table that is created in a partition-by-growth table space.

partition-expression
Specifies the key data over which the range is defined to determine the target data partition of the data.

column-name
Specifies the columns of the key. Each *column-name* must identify a column of the table. Do not specify more than 64 columns or the same column more than one time. The sum of length attributes of the columns must not be greater than 255 - *n*, where *n* is the number of columns that can contain null values. Do not specify a qualified column name.

A timestamp with time zone column (or a column with a distinct type that is based on the timestamp with time zone data type) can only be specified as the last column in a partitioning key.

Do not specify a column for *column-name* if the column is defined as follows:

- a LOB column (or a column with a distinct type that is based on a LOB data type)
- a BINARY column (or a column with a distinct type that is based on a BINARY data type)
- a VARBINARY column (or a column with a distinct type that is based on a VARBINARY data type)
- a DECFLOAT column (or a column with a distinct type that is based on a DECFLOAT data type)
- an XML column
- a row change timestamp column
- a Unicode column in an EBCDIC table

NULLS LAST
Specifies that null values are treated as positive infinity for purposes of comparison.

ASC
Puts the entries in ascending order by the column. ASC is the default.

DESC
Puts the entries in descending order by the column.

partition-element
Specifies ranges for a data partitioning key and the table space where rows of the table in the range will be stored.

PARTITION *integer*

integer is the physical number of a partition in the table space. A PARTITION clause must be specified for every partition of the table space. In this context, highest means highest in the sorting sequences of the columns. In a column defined as ascending (ASC), highest and lowest have their usual meanings. In a column defined as descending (DESC), the lowest actual value is highest in the sorting sequence.

ENDING AT (*constant***, MAXVALUE, or MINVALUE, ...)**

Defines the limit key for a partition boundary. Specify at least one value (*constant*, MAXVALUE, or MINVALUE) after ENDING AT in each PARTITION clause. You can use as many values as there are columns in the key. The concatenation of all values is the highest value of the key for ascending and the lowest for descending.

constant

Specifies a constant value with a data type that must conform to the rules for assigning that value to the column. If a string constant is longer or shorter than required by the length attribute of its column, the constant is either truncated or padded on the right to the required length. If the column is ascending, the padding character is X'FF'. If the column is descending, the padding character is X'00'. The precision and scale of a decimal constant must not be greater than the precision and scale of its corresponding column. A hexadecimal string constant (GX) cannot be specified.

MAXVALUE

Specifies a value greater than the maximum value for the limit key of a partition boundary (that is, all X'FF' regardless of whether the column is ascending or descending). If all of the columns in the partitioning key are ascending, a constant or the MINVALUE clause cannot be specified following MAXVALUE. After MAXVALUE is specified, all subsequent columns must be MAXVALUE.

MINVALUE

Specifies a value that is smaller than the minimum value for the limit key of a partition boundary (that is, all X'00' regardless of whether the column is ascending or descending). If all of the columns in the partitioning key are descending, a constant or the MAXVALUE clause cannot be specified following MINVALUE. After MINVALUE is specified, all subsequent columns must be MINVALUE.

The key values are subject to the following rules:

- The first value corresponds to the first column of the key, the second value to the second column, and so on. Using fewer values than there are columns in the key has the same effect as using the highest or lowest values for the omitted columns, depending on whether they are ascending or descending.
- The highest value of the key in any partition must be lower than the highest value of the key in the next partition for ascending cases.
- The values specified for the last partition are enforced. The value specified for the last partition is the highest value of the key that

can be placed in the table. Any key values greater than the value specified for the last partition are out of range.

- If the concatenation of all the values exceeds 255 bytes, only the first 255 bytes are considered.
- If a key includes a ROWID column or a column with a distinct type that is based on a ROWID data type, 17 bytes of the constant that is specified for the corresponding ROWID column are considered.
- If a null value is specified for the partitioning key and the key is ascending, an error is returned unless MAXVALUE is specified. If the key is descending, an error is returned unless MINVALUE is specified..

HASH SPACE *integer***K|M|G**
Specifies the amount of fixed hash space to preallocate for the partition that is associated with the *partition-element*. If HASH SPACE is omitted from the partition element, the **HASH SPACE** value from the **ORGANIZE BY** clause is used.

If **HASH SPACE** is not specified, each partition will use the **HASH SPACE** value specified in *organization-clause*.

The **HASH SPACE** keyword in *partition-element* must only be specified if *organization-clause* is also specified.

K Indicates that the integer value is to be multiplied by 1024 to specify the hash space size in bytes. The integer must be between 256 and 268435456.

M Indicates that the integer value is to be multiplied by 1048576 to specify the hash space size in bytes. The integer must be between 1 and 262144.

G Indicates that the integer value is to be multiplied by 1073741824 to specify the hash space size in bytes. The integer must be between 1 and 256 for a partition by range table and must be between 1 and 131072 for a non-partitioned table.

If a value greater than 4G is specified, the data sets for the table space are associated with a DFSMS data class that has been specified with extended format and extended addressability.

INCLUSIVE
Specifies that the specified range values are included in the data partition.

PARTITION BY SIZE
Specifies that the table is created in a partition-by-growth table space. If the IN clause is also specified, the IN clause must identify a partition-by-growth table space.

EVERY *integer* **G**
Specifies that the table is to be partitioned by growth, every *integer* G bytes. *integer* must not be greater than 256. If the IN clause identifies a table space, *integer* must be the same as the DSSIZE value that is in effect for the table space that will contain the table.

EDITPROC *program-name*
Identifies the user-written code that implements the edit procedure for the

table. The edit procedure must exist at the current server. The procedure is invoked during the execution of an SQL data change statement or LOAD and all row retrieval operations on the table.

An edit routine receives an entire table row, and can transform that row in any way. Also, it receives a transformed row and must change the row back to its original form.

You must not specify an edit routine for a table with a LOB column, or for an EBCDIC table with a Unicode column.

For information on writing an EDITPROC exit routine, see Edit procedures (DB2 Administration Guide).

WITH ROW ATTRIBUTES
Specifies that the edit procedure parameter list contains an address for the description of a row. WITH ROW ATTRIBUTES must not be specified for a table with an identity, LOB, XML, ROWID, or SECURITY LABEL column. WITH ROW ATTRIBUTES is the default. When WITH ROW ATTRIBUTES is specified, the column names in the table must not be longer than 18 EBCDIC SBCS characters in length.

WITHOUT ROW ATTRIBUTES
Specifies that the description of the row is not provided to the edit procedure. On entry to the edit procedure, the address for the row description in the parameter list contains a value of zero.

VALIDPROC *program-name*
Designates *program-name* as the validation exit routine for the table. Writing a validation exit routine is described in *DB2 Administration Guide*.

The validation routine can inhibit a load, insert, update, or delete operation on any row of the table: before the operation takes place, the procedure is passed the row. The values that are represented by any LOB or XML columns in the table are not passed to the validation routine. On an insert or update operation, if the table has a security label column and the user does not have write-down privilege, the user's security label value is passed to the validation routine as the value of the column. After examining the row, the procedure returns a value that indicates whether the operation should proceed. A typical use is to impose restrictions on the values that can appear in various columns.

A table can have only one validation procedure at a time. In an ALTER TABLE statement, you can designate a replacement procedure or discontinue the use of a validation procedure.

If you omit VALIDPROC, the table has no validation routine.

You must not specify a validation routine for an EBCDIC table with a Unicode column.

AUDIT
Identifies the types of access to this table that causes auditing to be performed. For information about audit trace classes, see *DB2 Administration Guide*.

If a materialized query table is refreshed with the REFRESH TABLE statement, the auditing also occurs during the REFRESH TABLE operation. **AUDIT** works as usual for LOAD and SQL data change operations on a user-maintained materialized query table.

NONE
Specifies that no auditing is to be done when this table is accessed. This is the default.

CHANGES
Specifies that auditing is to be done when the table is accessed during the first insert, update, or delete operation. However, the auditing is done only if the appropriate audit trace class is active.

ALL
Specifies that auditing is to be done when the table is accessed during the first operation of any kind performed by a utility or application process. However, the auditing is done only if the appropriate audit trace class is active and the access is not performed with COPY, RECOVER, REPAIR, or any stand-alone utility.

If the table is subsequently altered with an ALTER TABLE statement, the ALTER TABLE statement is audited for successful and failed attempts in the following cases, if the appropriate audit trace class is active:

- **AUDIT** attribute is changed to **NONE**, **CHANGES**, or **ALL** on an audited or non-audited table.
- **AUDIT CHANGES** or **AUDIT ALL** is in effect.

OBID *integer*
Identifies the OBID to be used for this table. An OBID is the identifier for an object's internal descriptor. The integer must be greater than 1 and must not identify an existing or previously used OBID of the database. If you omit OBID, DB2 generates a value.

The following statement retrieves the value of OBID:

```
SELECT OBID
  FROM SYSIBM.SYSTABLES
  WHERE CREATOR = 'ccc' AND NAME = 'nnn';
```

Here, *nnn* is the table name and *ccc* is the creator of the table.

DATA CAPTURE
Specifies whether the logging of the following actions on the table is augmented by additional information:

- SQL data change operations
- Adding columns (using the ADD COLUMN clause of the ALTER TABLE statement)
- Changing columns (using the ALTER COLUMN clause of the ALTER TABLE statement)

For guidance on intended uses of the expanded log records, see:

- The description of data propagation to IMS in *IMS DataPropagator: An Introduction*
- The instructions for using Remote Recovery Data Facility (RRDF) in *Remote Recovery Data Facility Program Description and Operations*
- The instructions for reading log records in *DB2 Administration Guide*

If a materialized query table is refreshed with the REFRESH TABLE statement, the logging of the augmented information occurs during the REFRESH TABLE operation. DATA CAPTURE works as usual for insert, update, and delete operations on a user-maintained materialized query table.

NONE
Do not record additional information to the log. This is the default.

CHANGES
Write additional data about SQL updates to the log. Information about the

values that are represented by any LOB or XML columns is not available. Do not specify DATA CAPTURE CHANGES for tables that reside in table spaces that specify NOT LOGGED.

CCSID *encoding-scheme*
Specifies the encoding scheme for string data stored in the table. If the IN clause is specified with a table space, the value must agree with the encoding scheme that is already in use for the specified table space. The specific CCSIDs for SBCS, mixed, and graphic data are determined by the table space or database specified in the IN clause. If the IN clause is not specified, the value specified is used for the table being created as well as for the table space that DB2 implicitly creates. The specific CCSIDs for SBCS, mixed, and graphic data are determined by the default CCSIDs for the server for the specified encoding scheme. The valid values are ASCII, EBCDIC, and UNICODE.

If the CCSID clause is not specified, the encoding scheme for the table depends on the IN clause:

- If the IN clause is specified, the encoding scheme already in use for the table space or database specified in the IN clause is used.
- If the IN clause is not specified, the encoding scheme of the new table is the same as the scheme for the table that is specified in the LIKE clause.

If the CCSID clause is specified for a materialized query table, the encoding scheme specified in the clause must be the same as the scheme for the result CCSID of the fullselect. The CCSID must also be the same as the CCSID of the table space for the table being created.

VOLATILE or NOT VOLATILE
Specifies how DB2 is to choose access to the table.

VOLATILE
Specifies that index access should be used on this table whenever possible for SQL operations.

One instance in which you might want to use VOLATILE is for a table whose size can vary greatly. If statistics are taken when the table is empty or has only a few rows, those statistics might not be appropriate when the table has many rows. Another instance in which you might want to use VOLATILE is for a table that contains groups of rows, as defined by the primary key on the table. All but the last column of the primary key of such a table indicate the group to which a given row belongs. The last column of the primary key is the sequence number indicating the order in which the rows are to be read from the group. VOLATILE maximizes concurrency of operations on rows within each group, since rows are usually accessed in the same order for each operation.

NOT VOLATILE
Specifies that SQL access to this table should be based on the current statistics. NOT VOLATILE is the default.

CARDINALITY
An optional keyword that currently has no effect, but that is provided for DB2 family compatibility.

LOGGED or NOT LOGGED
Specifies whether changes that are made to the data in the implicitly created table space are recorded in the log. This parameter applies to the implicitly created table space and to all indexes of this table. XML table spaces and indexes associated with the XML table spaces inherit the logging attribute from

the associated base table space. Auxiliary indexes also inherit the logging attribute from the associated base table space.

Do not specify **LOGGED** or **NOT LOGGED** if the table space name is specified by using the **IN** *table-space-name* clause.

LOGGED
Specifies that changes that are made to the data in the implicitly created table space are recorded in the log.

LOGGED is the default.

NOT LOGGED
Specifies that changes that are made to data in the implicitly created table space are not recorded in the log.

NOT LOGGED prevents undo and redo information from being recorded in the log. However, control information for the implicitly created table space will continue to be recorded in the log.

COMPRESS YES or COMPRESS NO
Specifies whether data compression applies to the rows of the implicitly created table space. The default is specified by the value of the subsystem parameter USE DATA COMPRESSION.

Do not specify **COMPRESS YES** or **COMPRESS NO** if the table space name is specified by using the **IN** *table-space-name* clause.

COMPRESS YES
Specifies that data compression applies to the rows of the implicitly created table space. The rows are not compressed until the LOAD or REORG utility is run on the table in the implicitly created table space.

COMPRESS NO
Specifies that data compression is not used for the rows of the implicitly created table space.

APPEND NO or APPEND YES
Specifies whether append processing is used for the table. The APPEND clause must not be specified for a table that is created in a work file table space.

NO Specifies that append processing is not used for the table. For insert and LOAD operations, DB2 will attempt to place data rows in a well clustered manner with respect to the value in the row's cluster key column.

NO is the default.

YES
Specifies that data rows are to be placed into the table by disregarding the clustering during insert and LOAD operations.

DSSIZE *integer* **G**
Specifies the maximum size for the implicitly created partition-by-growth or range-partitioned universal table space. This value is only applied to the implicitly created base table space, not to any associated implicitly created XML or LOB table spaces.

Do not specify **DSSIZE** *integer* **G** if any of the following conditions are true:

- The table space name is specified by using the **IN** *table-space-name* clause.
- The **PARTITION BY** clause includes the **EVERY** *integer-constant* **G** clause.

The default is specified by the value of the subsystem parameter MAXIMUM PARTITION SIZE.

For more detailed information about the DSSIZE clause, refer to "CREATE TABLESPACE" on page 1455.

BUFFERPOOL *bpname*
Specifies the buffer pool be use for the implicitly created table space and determines the page size of the table space. For 4KB, 8KB, 16KB and 32KB page buffer pools, the page sizes are 4 KB, 8 KB, 16 KB, and 32 KB, respectively.

bpname must identify an activated buffer pool. The privilege set must include SYSADM authority, SYSCTRL authority, or the USE privilege on the buffer pool.

Do not specify **BUFFERPOOL** *bpname* if the table space name is specified by using the **IN** *table-space-name* clause.

If you do not specify the BUFFERPOOL clause for an implicitly created table space, the default buffer pool of the database is used unless the record length exceeds the page size. If the record length exceeds the page size, do not specify the BUFFERPOOL clause. In this case, DB2 will choose a suitable buffer pool for the implicitly table space.

Refer to "Naming conventions" on page 57 for more information about *bpname*.

MEMBER CLUSTER
Specifies that data that is inserted by an insert operation is not clustered by the implicit clustering index (the first index) or the explicit clustering index. DB2 places the data in the implicitly created table space based on available space.

Do not specify **MEMBER CLUSTER** if the table space name is specified by using the **IN** *table-space-name* clause.

TRACKMOD YES or TRACKMOD NO
Specifies whether DB2 tracks modified pages in the space map pages of the implicitly created table space. The default is specified by the value of the subsystem parameter TRACK MODIFIED PAGES.

Do not specify **TRACKMOD YES** or **TRACKMOD NO** if the table space name is specified by using the **IN** *table-space-name* clause.

TRACKMOD YES
Changed pages are tracked in the space map pages to help improve performance of incremental image copies.

TRACKMOD NO
Changed pages are not tracked in the space map pages. DB2 uses the LRSN value in each page to determine whether a page has been changed.

materialized-query-definition
Specifies that the column definitions of the materialized query table are based on the result of a fullselect. If *materialized-query-table-options* are specified, the REFRESH TABLE statement can be used to populate the table with the results of the fullselect.

column-name
Names the columns in the table. If a list of column names is specified, it must consist of as many names as there are columns in the result table of the fullselect. Each *column-name* must be unique and unqualified. If a list of column names is not specified, the columns of the table inherit the names of the columns of the result table of the fullselect.

A list of column names must be specified if the result table of the fullselect has duplicate column names or an unnamed column. An unnamed column

is a column derived from a constant, function, expression, or set operation that is not named using the AS clause of the select list.

AS *(fullselect)*
Specifies that the table definition is based on the column definitions from the result of a query expression. The use of AS *(fullselect)* is an implicit definition of *n* columns for the table, where *n* is the number of columns that would result from the *fullselect*. The columns of the new table are defined by the columns that result from the *fullselect*. Every select list element must have a unique name. The AS clause can be used in the *select-clause* to provide unique names.

The implicit definition includes the column name, data type, length, precision, scale, and nullability characteristic of each of the result columns of *fullselect*. Other column attributes, such as DEFAULT, IDENTITY, and unique constraints, are not inherited from the *fullselect*. A column of the new table that corresponds to an implicitly hidden column of a base table referenced in the *fullselect* is not considered hidden in the new table. The generated column attributes are not inherited from the *fullselect*. That is, the new column of the materialized query table is not considered as a generated column. A FIELDPROC is inherited for a column if the corresponding select item of the *fullselect* is a column that can be directly mapped to a column of a base table or a view in the FROM clause of the *fullselect*. The materialized query table contains a security label column if only one table in the *fullselect* contains a security label column and the primary authorization ID of the statement has a valid security label.

Authorization for creating materialized query tables:
The owner of the table being created must have the SELECT privilege on the tables or views referenced in the fullselect, or the privilege set must include SYSADM or DBADM authority for the database in which the tables of the fullselect reside. Having SELECT privilege means that the owner has at least one of the following authorizations:

- Ownership of the tables or views referenced in the fullselect
- The SELECT privilege on the tables and views referenced in the fullselect
- SYSADM authority
- DBADM authority for the database in which the tables of the fullselect reside

The rules for establishing the qualifiers for names used in the fullselect are the same as the rules used to establish the qualifiers for *table-name*.

The following restrictions apply when creating materialized query tables. When *fullselect* does not satisfy the restrictions, an error occurs:

General restrictions: The following restrictions apply:

- The length of each result column of the fullselect must not be 0.
- The fullselect cannot contain a column of a LOB or XML data type.
- No more than one table in the fullselect can contain a security label column.
- The fullselect must not contain a period specification.
- The object that is specified in the FROM clause of the fullselect cannot be a view with columns of length 0.

- The fullselect cannot contain a reference to a created global temporary table, a declared global temporary table, or another materialized query table.
- The fullselect cannot directly or indirectly reference a base table that has been activated for the row or column access control or a base table for which a row permission or a column mask has been defined.
- The fullselect must not refer to host variables or include parameter markers.
- The fullselect must not refer to global variables.

Additional restrictions when ENABLE QUERY OPTIMIZATION is in effect:

- The fullselect must be a subselect.
- The outermost SELECT list of the subselect must not reference data that is encoded with different CCSID sets.
- The subselect cannot include the following:
  - A special register
  - A scalar fullselect
  - A row change timestamp column
  - A ROW CHANGE expression
  - An expression for which implicit time zone values apply (for example, cast a timestamp to a timestamp with time zone)
  - The RAND built-in function
  - The RID built-in function
  - A user-defined scalar or table function that is not deterministic or that has external actions
  - Any predicates that include a subquery
  - A row expression predicate
  - A join using the INNER JOIN syntax, or an outer join
  - A lateral correlation
  - A nested table expression or view that requires temporary materialization
  - A direct or indirect reference to a table that uses activated row or column access controls, or a table for which row or column access controls have been defined.
  - A FETCH FIRST clause
  - A reference to a global variable
  - A collection-derived table (UNNEST)
- If a table with a security label is referenced, the security label column must be referenced in the outer select list of the subselect.
- If the subselect references a view, the fullselect in the view definition must satisfy all other restrictions.

refreshable-table-options

Specifies the options for a refreshable materialized query table. The ORDER BY clause is allowed, but it is used only by REFRESH. The ORDER BY clause can improve the locality of reference of data in the materialized query table.

DATA INITIALLY DEFERRED

Specifies that the data is not inserted into the materialized query table

when it is created. Use the REFRESH TABLE statement to populate the materialized query table, or use the INSERT statement to insert data into a user-maintained materialized query table.

REFRESH DEFERRED
Specifies that the data in the table can be refreshed at any time using the REFRESH TABLE statement. The data in the table only reflects the result of the query as a snapshot at the time when the REFRESH TABLE statement is processed or when it was last updated for a user-maintained materialized query table.

MAINTAINED BY SYSTEM or MAINTAINED BY USER
Specifies how the data in the materialized query table is maintained.

MAINTAINED BY SYSTEM
Specifies that the materialized query table is maintained by the system. Only the REFRESH statement is allowed on the table. This is the default.

MAINTAINED BY USER
Specifies that the materialized query table is maintained by the user, who can use the LOAD utility, an SQL data change statement, a SELECT from data change statement, or REFRESH TABLE SQL statements on the table.

ENABLE QUERY OPTIMIZATION or DISABLE QUERY OPTIMIZATION
Specifies whether this materialized query table can be used for optimization.

ENABLE QUERY OPTIMIZATION
Specifies that the materialized query table can be used for query optimization. If the fullselect specified does not satisfy the restrictions for query optimization, an error occurs.

ENABLE QUERY OPTIMIZATION is the default.

The fullselect must not contain a period specification.

DISABLE QUERY OPTIMIZATION
Specifies that the materialized query table cannot be used for query optimization. The table can still be queried directly.

Notes

Owner privileges:
The owner of the table has all table privileges (see "GRANT (table or view privileges)" on page 1721) with the ability to grant these privileges to others. For more information about ownership of the object, see "Authorization, privileges, permissions, masks, and object ownership" on page 70.

Table design:
Designing tables is part of the process of database design. For information on design, see *Introduction to DB2 for z/OS*.

Creating a table in a segmented table space:
A table cannot be created in a segmented table space if any of the following conditions are true:

- The available space in the data set is less than the segment size specified for the table space, and
- The data set cannot be extended.

Creating a table with graphic and mixed data columns:

You cannot create an ASCII or EBCDIC table with a GRAPHIC, VARGRAPHIC, or DBCLOB column or a CHAR, VARCHAR, or CLOB column defined as FOR MIXED DATA when the setting for installation option MIXED DATA is NO.

Creating a table with distinct type columns based on LOB, ROWID, and DECFLOAT columns:

Because a distinct type is subject to the same restrictions as its source type, all the syntactic rules that apply to LOB columns (CLOB, DBCLOB, and BLOB), ROWID columns, and DECFLOAT columns apply to distinct type columns that are based on LOBs, row IDs, and DECFLOATs. For example, a table cannot have both an explicitly defined ROWID column and a column with a distinct type that is based on a row ID.

Tables with inline LOB columns:

If the 32K page size is exceeded for a table in an universal table space, DB2 recalculates the record size using 0 as the inline length for LOB columns that do not specify the INLINE LENGTH clause. After the recalculation, if the 32K page size is still exceeded, the CREATE TABLE statement returns an error.

You cannot create a table with an inline LOB column in a table space that has basic row format.

Creating a table with LOB columns:

A table with a LOB column (CLOB, DBCLOB, or BLOB) must also have a ROWID column and one or more auxiliary tables. When you create the table, DB2 implicitly generates a ROWID column for you. This is called an *implicitly hidden ROWID column*, and DB2:

- Creates the column with a name of DB2_GENERATED_ROWID_FOR_LOBS*nn*.

  DB2 appends *nn* only if the column name already exists in the table, replacing *nn* with 00 and incrementing by 1 until the name is unique within the row.
- Defines the column as GENERATED ALWAYS.
- Appends the implicitly hidden ROWID column to the end of the row after all the other explicitly defined columns.

For example, assume that DB2 generated an implicitly hidden ROWID column named DB2_GENERATED_ROWID_FOR_LOBS for table MYTABLE. The result table for a SELECT * statement for table MYTABLE would not contain that ROWID column. However, the result table for SELECT COL1, DB2_GENERATED_ROWID_FOR_LOBS would include the implicitly hidden ROWID column.

If the MIXED DATA subsystem parameter is set to yes, and a lowercase or mixed case hexadecimal constant is specified as the default value for a LOB column, the CREATE TABLE statement returns an error.

The definition of the table is marked incomplete until an auxiliary table is created in a LOB table space for each LOB column in the base table and index is created on each auxiliary table. The auxiliary table stores the actual values of a LOB column. If you create a table with a LOB column in a partitioned table space, there must be one auxiliary table defined for each partition of the base table space.

Unless DB2 implicitly creates the LOB table space, auxiliary table, and index on the auxiliary table for each LOB column in the base table, you

need to create these objects using the CREATE TABLESPACE, CREATE AUXILIARY TABLE, and CREATE INDEX statements.

If the table space that contains the table is explicitly created and the value of the CURRENT RULES special register is 'STD' when the CREATE TABLE statement is processed, or the table space that contains the table is implicitly created, DB2 implicitly creates the LOB table space, auxiliary table, and index on the auxiliary table for each LOB column in the base table.

The privilege set must include the following privileges:

- The USE privilege on the buffer pool and the storage group that is used by the XML objects
- If the base table space is explicitly created, CREATETS is also required on the database that contains the table (DSNDB04 if the database is implicitly created)

DB2 chooses the names of implicitly created objects using these conventions:

LOB table space
Name is 8 characters long, consisting of an 'L' followed by 7 random characters.

auxiliary table
Name is 18 characters long. The first five characters of the name are the first five characters of the name of the base table. The second five characters are the first five characters of the name of the LOB column. The last eight characters are randomly generated. If a base table name or a LOB column name is less than five characters, DB2 adds underscore characters to the name to pad it to a length of five characters.

index on the auxiliary table
Name is 18 characters long. The first character of the name is an 'I'. The next ten characters are the first ten characters of the name of the auxiliary table. The last seven characters are randomly generated. The index has the COPY NO attribute.

The other attributes of these implicitly created objects are those that would have been created by their respective CREATE statements with all optional clauses omitted, with the following exceptions:

- The database name is the database name of the base table.
- If the LOB table space is implicitly created, the buffer pool is determined by the DEFAULT BUFFER POOL FOR USER LOB DATA fields of installation panel DSNTIP1. The appropriate USE privilege is required on that buffer pool.

Utility REPORT TABLESPACESET identifies the LOB table spaces that DB2 implicitly created.

Creating a table with an XML column:
When a table is created with an XML column, an XML table space, XML table, and a node ID index and document ID index are implicitly created.

The privilege set must include the following privileges:

- The USE privilege on the buffer pool and the storage group that is used by the XML objects

- If the base table space is explicitly created, CREATETS is also required on the database that contains the table (DSNDB04 if the database is implicitly created)

The buffer pool for the XML table space is determined by the DEFAULT BUFFER POOL FOR USER XML DATA fields of installation panel DSNTIP1. The appropriate USE privilege is required on that buffer pool.

The XML table space will have a larger DSSIZE than the base table space if the base table space is partitioned by range. If the base table space is partitioned by growth, the default DSSIZE of 4GB will be used for the XML table space. The DSSIZE for an XML table space that is associated with a partitioned by range base table space is determined as follows.

Table 118. Default DSSIZE for XML table spaces, given base table space DSSIZE and page size

Base table space DSSIZE	4KB base page size	8KB base page size	16KB base page size	32KB base page size
1GB - 4GB	4GB	4GB	4GB	4GB
8GB	32GB	16GB	16GB	16GB
16GB	64GB	32GB	16GB	16GB
32GB	64GB	64GB	32GB	16GB
64GB	64GB	64GB	64GB	32GB
128GB	256GB	256GB	128GB	64GB
256GB	256GB	256GB	256GB	128GB

For example: for a base table space that has a DSSIZE of 8GB and a page size of 8KB, the XML table space will have a DSSIZE of 16GB.

Naming convention for implicitly created XML objects:

Implicitly created XML table spaces names will be X*yyynnnn*, where *yyy* is derived from the first three bytes of the base table name (if the name is shorter than 3, *yyy* is padded with X). *nnnn* is a numeric string that will start at 0000 and be incremented by 1 until a unique number is found.

Implicitly created XML table names will be X*yyyyyyyyyyyyyyyyyyynnn*, where *yyyyyyyyyyyyyyyyyy* is the first 18 UTF-8 bytes of the base table name or of the entire name if it is less than 18. *nnn* will only be appended if the name already exists in the table. If the name already exists, *nnn* will be replaced with 000 and will be incremented by 1 until the name is unique.

Implicitly created document ID index names will be I_DOCID*yyyyyyyyyyyyyyyyyyynnn*, where *yyyyyyyyyyyyyyyyyy* is the first 18 UTF-8 bytes of the base table name or the entire name if it is less than 18. *nnn* will only be appended if the index already exists in the table. If the index already exists, *nnn* will be replaced with 000 and will be incremented by 1 until the name is unique.

Implicitly created node ID index names will be I_NODEID*yyyyyyyyyyyyyyyyyyynnn*, where *yyyyyyyyyyyyyyyyyy* is the first 18 UTF-8 bytes of the XML table name or the entire name if it is less than 18. *nnn* will only be appended if the index already exists in the table. If the index already exists, *nnn* will be replaced with 000 and will be incremented by 1 until the name is unique.

Creating a table with an identity column:

When a table has an identity column, DB2 can automatically generate sequential numeric values for the column as rows are inserted into the table. Thus, identity columns are ideal for primary keys. Identity columns and ROWID columns are similar in that both types of columns contain values that DB2 generates. ROWID columns are used in large object (LOB) table spaces and can be useful in direct-row access. ROWID columns contain values of the ROWID data type, which returns a 40 byte VARCHAR value that is not regularly ascending or descending. ROWID data values are therefore not well suited to many application uses, such as generating employee numbers or product numbers. For data that is not LOB data and that does not require direct-row access, identity columns are usually a better approach, because identity columns contain existing numeric data types and can be used in a wide variety of uses for which ROWID values would not be suitable.

When a table is recovered to a point-in-time, it is possible that a large gap in the generated values for the identity column might result. For example, assume a table has an identity column that has an incremental value of 1 and that the last generated value at time T1 was 100 and DB2 subsequently generates values up to 1000. Now, assume that the table space is recovered back to time T1. The generated value of the identity column for the next row that is inserted after the recovery completes will be 1001, leaving a gap from 100 to 1001 in the values of the identity column.

If you want to ensure that an identity column has unique values, create a unique index on the column.

Creating a table with a LONG VARCHAR or LONG VARGRAPHIC column:

Although the syntax LONG VARCHAR and LONG VARGRAPHIC is allowed for compatibility with previous releases of DB2, its use is not encouraged. VARCHAR(*integer*) and VARGRAPHIC(*integer*) is the recommended syntax, because after the CREATE TABLE statement is processed, DB2 considers a LONG VARCHAR column to be VARCHAR and a LONG VARGRAPHIC column to be VARGRAPHIC.

When a column is defined using the LONG VARCHAR or LONG VARGRAPHIC syntax, DB2 determines the length attribute of the column. You can use the following information, which is provided for existing applications that require the use of the LONG VARCHAR or LONGVARGRAPHIC syntax, to calculate the byte count and the character count of the column.

To calculate the byte count, use this formula:

2*(INTEGER((INTEGER((*m*-*i*-*k*)/*j*))/2))

Where:

m Is the maximum row size (8 less than the maximum record size)

i Is the sum of the byte counts of all columns in the table that are not LONG VARCHAR or LONG VARGRAPHIC

j is the number of LONG VARCHAR and LONG VARGRAPHIC columns in the table

k k is the number of LONG VARCHAR and LONG VARGRAPHIC columns that allow nulls

To find the character count:

1. Find the byte count.
2. Subtract 2.
3. If the data type is LONG VARGRAPHIC, divide the result by 2. If the result is not an integer, drop the fractional part.

Considerations for a Unicode column in an EBCDIC table:

A column in an EBCDIC table can be defined for data encoded as Unicode UTF8 or UTF16. Specify the CCSID clause as the last part of the built-in-type specification to define a Unicode column. The CCSID clause can only be specified with the VARCHAR or VARGRAPHIC keywords as indicated, where *n* is the declared length:

Unicode encoding	Valid column definition
UTF8	VARCHAR(*n*) CCSID 1208
UTF16	VARGRAPHIC(*n*) CCSID 1200

If the IN DATABASE clause is specified:

If you specify IN DATABASE (either explicitly or by default), but do not specify a table space, a table space is implicitly created in *database-name*. The name of the table space is derived from the table name. The qualifier of the table space is the same as the qualifier of the table. The owner of the table space is SYSIBM.

If range-partitioning is not specified, the implicitly created table space is a partition-by-growth table space with MAXPARTITIONS 256, SEGSIZE 32, and DSSIZE 4G.

If range-partitioning is specified, the table space will be partitioned based on the number of parts specified on the CREATE TABLE statement with defaults of SEGSIZE 32, LOCKSIZE ANY, and LOCKMAX SYSTEM.

If the IN clause is not specified:

If you do not specify the IN clause, the DB2 subsystem will implicitly create a table space as described previously, but the DB2 subsystem will also choose a database. The DB2 subsystem creates a name in the form of DSN*nnnnn*, where *nnnnn* is between 00001 and the maximum value of the sequence SYSIBM.DSNSEQ_IMPLICITDB, which has a default of 10000, inclusive. The owner of the database is SYSIBM.

- If DSN*nnnnn* already exists and is an implicitly created database, the DB2 subsystem creates the table in that database.
- If DSN*nnnnn* does not exist, the DB2 subsystem creates a database with the name DSN*nnnnn*.

If DSN*nnnnn* cannot be created because of a deadlock, timeout, or resource unavailable condition, the DB2 subsystem increments *nnnnn* by one and tries the resultant database name. If the DB2 subsystem reaches the maximum value of the sequence SYSIBM.DSNSEQ_IMPLICITDB, and the corresponding database name is not available, the DB2 subsystem sets *nnnnn* to 00001 and tries the resultant database name. If the DB2 subsystem attempts to create the table a number of times that is equal to the maximum value of the sequence SYSIBM.DSNSEQ_IMPLICITDB without success, an error occurs.

Implicitly created table space attributes:

The attributes of the implicitly created table space can be changed by using the "ALTER TABLESPACE" on page 1074 statement.

Defining a system-period temporal table:

A system-period temporal table definition includes the following:

- A system period named SYSTEM_TIME which is defined using a row-begin column and a row-end column.
- A transaction-start-ID column.
- A system-period data versioning definition which includes the name of the associated history table.

To ensure that the history table cannot be implicitly dropped when a system-period temporal table is dropped, use the WITH RESTRICT ON DROP clause in the definition of the history table.

Defining an application-period temporal table:

An application-period temporal table definition includes an application period named BUSINESS_TIME. The application period is defined using a begin timestamp column and an end timestamp column.

Data change operations on an application-period temporal table might result in an automatic insert of one or two additional rows when a row is updated or deleted. When an update or delete of a row in an application-period temporal table is specified for a portion of the period that is represented by that row, the row is updated or deleted and one or two rows are automatically inserted to represent the portion of the row that is not changed. New values are generated for each generated column in an application-period temporal table for each row that is automatically inserted as a result of an update or delete operation on the table. If a generated column is defined as part of a unique or primary key, parent key in a referential constraint, or unique index, it is possible that an automatic insert will violate a constraint or index, in which case an error is returned.

Bitemporal tables:

A table that is defined for system-period data versioning and contains a BUSINESS_TIME period is referred to as a bitemporal table.

Considerations for transaction-start-ID columns:

A transaction-start-ID column contains a null value if the column allows null values. A row-begin column which is unique from other row-begin column values that are generated for other transactions exists with the transaction-start-ID column. Given that the column might contain null values, consider using one of the following methods when retrieving a value from the column:

```
COALESCE ( transaction_start_id_col, row_begin_col)
CASE WHEN transaction_start_id_col IS NOT NULL
 THEN transaction_start_id_col
  ELSE row_begin_col
END
```

If the IN clause is specified with ORGANIZE BY HASH:

If you specify IN DATABASE (either explicitly or by default), and ORGANIZE BY HASH, DB2 will calculate an optimum buffer pool for hash organization based on the definition of the table and validate the calculated buffer pool with the buffer pool of the explicitly created table space. If the buffer pool sizes are different, DB2 will return an error.

If the table is in a range-partitioned universal table space, the DSSIZE value for the table space must be large enough to fit the HASH SPACE specification for each partition.

If the table is in a partition-by-growth table space, the total space calculated from the DSSIZE and MAXPARTITIONS values for the table space must be large enough for the implicitly or explicitly specified HASH SPACE.

If the IN clause is not specified with ORGANIZE BY HASH:
If you do not specify IN DATABASE (either explicitly or by default), DB2 will use the default DSSIZE of 4G for each partition for a range-partition universal table space or use the value that is specified in the partitioning clause. The hash space value that is specified on CREATE TABLE will be validated, per part, to ensure that the specified DSSIZE is adequate. If the DSSIZE is not adequate, an error will be returned.

If the maximum number of partitions needed for the specified hash space is more than the maximum number of partitions allowed, DB2 to will return an error.

If the selected buffer pool is not available, a error will be returned.

Creating a table with hash organization and LOB columns:
If the table space is a partition-by-growth universal table space, DB2 will preallocate as many partitions as needed depending on the value specified for HASH SPACE. If DB2 needs to implicitly create the LOB object in a new partition, the privilege set for the implicitly created LOB objects must include the USE privilege on the buffer pool for the LOB table space.

Hash space and DB2 page size:
If the specified hash space is less than or equal to 64 MB (the DB2 default), DB2 will add extra space for DB2 system pages. If the specified hash space is greater than 64 MB, DB2 will use part of the hash space for DB2 system pages. The amount of space needed for DB2 system pages depends on SEGSIZE and PAGESIZE. The larger the SEGSIZE and/or PAGESIZE becomes, the larger the requirement for DB2 system pages. DB2 can reserve up to 5 MB for system pages for the highest SEGSIZE value (64) and PAGESIZE value (32K).

Hash space and DSSIZE:
Depending on certain table space characteristics, DB2 needs to reserve space for the hash overflow area. Therefore, the amount of hash space cannot be equal to the DSSIZE value. The maximum amount of hash space that can be specified is approximately 20% less than the DSSIZE value. DB2 returns an error if the amount of hash space is too large. If the amount of hash space is too large, specify a larger value of DSSIZE, or decrease the amount of hash space.

Specifying APPEND with ORGANIZE BY HASH:
Append processing is not applicable to tables with hash organization since there is no key clustering in hash organization. For insert operations into tables with hash organization, DB2 will use the internal hash algorithm to determine the location of the row.

Restrictions for tables with hash organization:
Tables that use hash organization are subject to the following restrictions:

- A table that is defined to use hash organization cannot be created in a LOB table space or XML table space.
- ORGANIZE BY HASH must not be specified if the table space is defined with the MEMBER CLUSTER clause.

- The MAXROWS clause is applicable only to the hash overflow area of the table space for tables with hash organization. The fixed hash area of each page will contain as many rows as it can hold, up to a maximum of 255.
- The ORGANIZE BY HASH UNIQUE (*column-list*) clause is required when specifying HASH SPACE *integer* K|M|G in the *partition-element*. The *organization-clause* applies to the entire table and the *partition-element* clause applies at the partition level.
- DB2 will automatically create a hash overflow index when a table is created with hash organization.

Implicitly created table spaces:

If the table space is implicitly created, all of the following required system objects will also be implicitly created:

- The enforcing primary key index
- The enforcing unique key index
- Any necessary LOB table spaces, auxiliary table spaces, and auxiliary indexes
- The ROWID index (if the ROWID column is defined as GENERATED BY DEFAULT)

When DB2 implicitly creates a base table space for a table with LOB columns that can have inline LOBs, DB2 creates the base table space in reordered row format, regardless of the value of the RRF subsystem parameter.

When DB2 implicitly creates a table space for a table with hash organization, DB2 creates the table space in reordered row format, regardless of the value of the RRF subsystem parameter.

Implicitly created indexes:

When the PRIMARY KEY or UNIQUE clause is used in the CREATE TABLE statement and the CREATE TABLE statement is processed by the schema processor or the table space that contains the table is implicitly created, DB2 implicitly creates the unique indexes used to enforce the uniqueness of the primary or unique keys.

When a ROWID column is defined as GENERATED BY DEFAULT in the CREATE TABLE statement, and the CREATE TABLE statement is processed by SET CURRENT RULES = 'STD' or the table space that contains the table is implicitly created, DB2 implicitly creates the unique indexes used to enforce the uniqueness of the ROWID column.

The privilege set must include the USE privilege of the buffer pool.

Each index is created as if the following CREATE INDEX statement were issued:

```
CREATE UNIQUE INDEX xxx ON table-name (column1,...)
```

Where:

- *xxx* is the name of the index that DB2 generates.
- *table-name* is the name of the table specified in the CREATE TABLE statement.
- *(column1,...)* is the list of column names that were specified in the UNIQUE or PRIMARY KEY clause of the CREATE TABLE statement, or the column is a ROWID column that is defined as GENERATED BY DEFAULT.

For more information about the schema processor, see *DB2 Administration Guide*.

In addition, if the table space that contains the table is implicitly created, DB2 will check the DEFINE DATA SET subsystem parameter to determine whether to define the underlying data set for the index space of the implicitly created index on the base table.

If DEFINE DATA SET is NO, the index is created as if the following CREATE INDEX statement is issued:

```
CREATE UNIQUE INDEX xxx ON table-name (column1,...) DEFINE NO
```

Maximum record size:

The maximum record size of a table depends on the page size of the table space, whether the table space is organized for hash access, and whether the EDITPROC clause is specified, as shown in Table 119.

The initial page size of the table space is the size of its buffer, which is determined by the BUFFERPOOL clause that was explicitly or implicitly specified when the table space was created. When the record size reaches 90 percent of the maximum record size for the page size of the table space, the next largest page size is automatically used.

Table 119. Maximum record size, in bytes

	Page Size = 4KB	**Page Size = 8KB**	**Page Size = 16KB**	**Page Size = 32KB**
Non-hash table	4056	8138	16330	32714
Non-hash table with EDITPROC=YES	4046	8128	16320	32704
Hash table (hash home page)	3817	7899	16091	32475
Hash table with EDITPROC=YES (hash home page)	3807	7889	16081	32465

The maximum record size corresponds to the maximum length of a VARCHAR column if that column is the only column in the table.

If the table space that contains the table is implicitly created, the proper buffer pool size is chosen according to the actual record size. If the record size reaches 90% of the maximum record size for the page size of the table space, the next largest page size will be used. Table 120 shows 90% of the maximum record size:

Table 120. 90% of Maximum record size, in bytes

	Page Size = 4KB	**Page Size = 8KB**	**Page Size = 16KB**	**Page Size = 32KB**
Non-hash table	3650	7324	14697	29443
Non-hash table with EDITPROC=YES	3641	7315	14688	29434

Table 120. 90% of Maximum record size, in bytes (continued)

	Page Size = 4KB	Page Size = 8KB	Page Size = 16KB	Page Size = 32KB
Hash table (hash home page)	3435	7109	14482	29228
Hash table with EDITPROC=YES (hash home page)	3426	7100	14473	29219

Byte counts:

The sum of the byte counts of the columns must not exceed the maximum row size of the table. The maximum row size is eight less than the maximum record size.

For columns that do not allow null values, Table 121 gives the byte counts of columns by data type. For columns that allow null values, the byte count is one more than shown in the table.

Table 121. Byte counts of columns by data type

Data Type	Byte Count
INTEGER	4
SMALLINT	2
BIGINT	8
FLOAT(*n*)	If *n* is between 1 and 21, the byte count is 4. If *n* is between 22 and 53, the byte count is 8.
DECIMAL	INTEGER(*p*/2)+1, where *p* is the precision
DECFLOAT(16)	9
DECFLOAT(34)	17
CHAR(*n*)	*n*
VARCHAR(*n*)	*n*+2
CLOB	6
Inline CLOB	6 + inline byte count
GRAPHIC(*n*)	2*n*
VARGRAPHIC(*n*)	2*n*+2
DBCLOB	6
Inline DBCLOB	6 + (inline char count * 2)
BINARY(*n*)	*n*
VARBINARY(*n*)	*n*+2
BLOB	6
Inline BLOB	6 + inline byte count
DATE	4
TIME	3
TIMESTAMP(*p*) WITHOUT TIME ZONE	INTEGER((*p*+1)/2) + 7 where *p* is the precision
TIMESTAMP(*p*) WITH TIME ZONE	INTEGER((*p*+1)/2) + 9 where *p* is the precision

Table 121. Byte counts of columns by data type (continued)

Data Type	Byte Count
ROWID	19
distinct type	The length of the source data type upon which the distinct type was based

Creating a materialized query table:

If the fullselect in the CREATE TABLE statement contains a SELECT *, the select list of the subselect is determined at the time the materialized query table is created. In addition, any references to user-defined functions are resolved at the same time. The isolation level at the time when the CREATE TABLE statement is executed is the isolation level for the materialized query table. After a materialized query table is created, the REFRESH_TIME column of the row for the table in the SYSIBM.SYSVIEWS catalog table contains the default timestamp.

The owner of a materialized query table has all the table privileges with the grant option on the table irrespective of whether the owner has the necessary privileges on the base tables, views, functions, and sequences.

No unique constraints or unique indexes can be created for materialized query tables. Thus, a materialized query table cannot be a parent table in a referential constraint.

When you are creating user-maintained materialized query tables, you should create the materialized query table with query optimization disabled and then enable the table for query optimization after it is populated. Otherwise, DB2 might rewrite queries to use the empty materialized query table, and you will not get accurate results.

Considerations for implicitly hidden columns:

A column that is defined as implicitly hidden is not part of the result table of a query that specifies * in a SELECT list. However, an implicitly hidden column can be explicitly referenced in a query. For example, an implicitly hidden column can be referenced in the SELECT list or in a predicate in a query. Additionally, an implicitly hidden column can be explicitly referenced in a COMMENT, CREATE INDEX statement, ALTER TABLE statement, INSERT statement, MERGE statement, UPDATE statement, or RENAME statement. An implicitly hidden column can be referenced in a referential constraint. A REFERENCES clause that does not contain a column list refers implicitly to the primary key of the parent table. It is possible that the primary key of the parent table includes a column defined as implicitly hidden. Such a referential constraint is allowed.

If the SELECT list of the fullselect of a materialized query definition explicitly refers to an implicitly hidden column, that column will be part of the materialized query table.

If the SELECT list of the fullselect of a view definition (CREATE VIEW statement) explicitly refers to an implicitly hidden column, that column will be part of the view, however the view column is not considered 'hidden'.

Restrictions on field procedures, edit procedures, and validation exit procedures:

Field procedures, edit procedures, and validation exit procedures cannot be used on tables that have column names that are larger than 18 EBCDIC bytes. If you have tables that have field procedures or validation exit

procedures and you add a column where the column name is larger than 18 bytes, the field procedures and validation exit procedures for the table will be invalidated.

Consider using triggers to replace the functionality on field procedures, edit procedures, and validation exit procedures on tables where the column names are larger than 18 EBCDIC bytes.

Restrictions on SQL data change statements in the same unit of work as CREATE TABLE:
SQL data change statements cannot follow, in the same unit of work, CREATE TABLE statements that specify the PARTITION BY clause.

Creating a table while a utility runs:
You cannot use CREATE TABLE while a DB2 utility has control of the table space implicitly or explicitly specified by the IN clause.

Restrictions involving pending definition changes:
A CREATE TABLE statement is not allowed if there are pending changes to the definition of the table space, if the CREATE TABLE statement specifies a FOREIGN KEY clause that reference a column for which there are pending definition changes, or if the CREATE TABLE statement specifies a materialized query table definition that references a table for which there are pending definition changes.

Alternative syntax and synonyms:
To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following clauses:

- NOCACHE (single clause) as a synonym for NO CACHE
- NOCYCLE (single clause) as a synonym for NO CYCLE
- NOMINVALUE (single clause) as a synonym for NO MINVALUE
- NOMAXVALUE (single clause) as a synonym for NO MAXVALUE
- NOORDER (single clause) as a synonym for NO ORDER
- PART *integer* VALUES can be specified as an alternative to PARTITION *integer* ENDING AT.
- VALUES as a synonym for ENDING AT
- DEFINITION ONLY as a synonym for WITH NO DATA
- SUMMARY between CREATE and TABLE
- TIMEZONE can be specified as an alternative to TIME ZONE.

Examples

Example 1: Create a table named DSN8B10.DEPT in the table space DSN8S11D of the database DSN8D11A. Name the five columns DEPTNO, DEPTNAME, MGRNO, ADMRDEPT, and LOCATION, allowing only MGRNO and LOCATION to contain nulls, and designating DEPTNO as the only column in the primary key. All five columns hold character string data. Assuming a value of NO for the field MIXED DATA on installation panel DSNTIPF, all five columns have the subtype SBCS.

```
CREATE TABLE DSN8B10.DEPT
  (DEPTNO   CHAR(3)     NOT NULL,
   DEPTNAME VARCHAR(36) NOT NULL,
   MGRNO    CHAR(6)             ,
   ADMRDEPT CHAR(3)     NOT NULL,
   LOCATION CHAR(16)            ,
   PRIMARY KEY(DEPTNO)          )
  IN DSN8D11A.DSN8S11D;
```

Example 2: Create a table named DSN8B10.PROJ in an implicitly created table space of the database DSN8D11A. Assign the table a validation procedure named DSN8EAPR.

```
CREATE TABLE DSN8B10.PROJ
  (PROJNO   CHAR(6)      NOT NULL,
   PROJNAME VARCHAR(24)  NOT NULL,
   DEPTNO   CHAR(3)      NOT NULL,
   RESPEMP  CHAR(6)      NOT NULL,
   PRSTAFF  DECIMAL(5,2)         ,
   PRSTDATE DATE                 ,
   PRENDATE DATE                 ,
   MAJPROJ  CHAR(6)      NOT NULL)
  IN DATABASE DSN8D11A
  VALIDPROC DSN8EAPR;
```

Example 3: Assume that table PROJECT has a non-primary unique key that consists of columns DEPTNO and RESPEMP (the department number and employee responsible for a project). Create a project activity table named ACTIVITY with a foreign key on that unique key.

```
CREATE TABLE ACTIVITY
  (PROJNO   CHAR(6)      NOT NULL,
   ACTNO    SMALLINT     NOT NULL,
   ACTDEPT  CHAR(3)      NOT NULL,
   ACTOWNER CHAR(6)      NOT NULL,
   ACSTAFF  DECIMAL(5,2)         ,
   ACSTDATE DATE         NOT NULL,
   ACENDATE DATE                 ,
   FOREIGN KEY (ACTDEPT,ACTOWNER)
      REFERENCES PROJECT (DEPTNO,RESPEMP) ON DELETE RESTRICT)
  IN DSN8D11A.DSN8S11D;
```

Example 4: Create an employee photo and resume table EMP_PHOTO_RESUME that complements the sample employee table. The table contains a photo and resume for each employee. Put the table in table space DSN8D11A.DSN8S11E. Let DB2 always generate the values for the ROWID column.

```
CREATE TABLE DSN8B10.EMP_PHOTO_RESUME
  (EMPNO      CHAR(6)     NOT NULL,
   EMP_ROWID  ROWID NOT NULL GENERATED ALWAYS,
   EMP_PHOTO  BLOB(110K),
   RESUME     CLOB(5K),
   PRIMARY KEY (EMPNO))
  IN DSN8D11A.DSN8S11E
  CCSID EBCDIC;
```

Example 5: Create an EMPLOYEE table with an identity column named EMPNO. Define the identity column so that DB2 will always generate the values for the column. Use the default value, which is 1, for the first value that should be assigned and for the incremental difference between the subsequently generated consecutive numbers.

```
CREATE TABLE EMPLOYEE
  (EMPNO      INTEGER GENERATED ALWAYS AS IDENTITY,
   ID         SMALLINT,
   NAME       CHAR(30),
   SALARY     DECIMAL(5,2),
   DEPTNO     SMALLINT)
  IN DSN8D11A.DSN8S11D;
```

Example 6: Assume a very large transaction table named TRANS contains one row for each transaction processed by a company. The table is defined with many columns. Create a materialized query table for the TRANS table that contain daily summary data for the date and amount of a transaction.

```
CREATE TABLE STRANS AS
  (SELECT YEAR AS SYEAR, MONTH AS SMONTH, DAY AS SDAY, SUM(AMOUNT) AS SSUM
   FROM TRANS
   GROUP BY YEAR, MONTH, DAY)
   DATA INITIALLY DEFERRED REFRESH DEFERRED;
```

Example 7: The following example creates a table in a partition-by-growth table space and includes the APPEND option:

```
CREATE TABLE TS01TB
     (C1 SMALLINT,
      C2 DECIMAL(9,2),
      C3 CHAR(4))
  APPEND YES
  IN TS01DB.TS01TS;
```

Example 8: The following example creates a table in a partition-by-growth table space where the table space is implicitly created:

```
CREATE TABLE TS02TB
     (C1 SMALLINT,
      C2 DECIMAL(9,2),
      C3 CHAR(4))
   PARTITION BY SIZE EVERY 4G
   IN DATABASE DSNDB04;
```

Example 9: Create a table, EMP_INFO, that contains a phone number and address for each employee. Include a row change timestamp column in the table to track the modification of employee information:

```
CREATE TABLE EMP_INFO
     (EMPNO CHAR(6) NOT NULL,
      EMP_INFOCHANGE NOT NULL
         GENERATED ALWAYS FOR EACH ROW ON UPDATE
         AS ROW CHANGE TIMESTAMP,
      EMP_ADDRESS VARCHAR(300),
      EMP_PHONENO CHAR(4),
      PRIMARY KEY (EMPNO));
```

Example 10: Create a table, TB01, that uses a range partitioning scheme with a segment size of 4 and 4 partitions.

```
CREATE TABLE TB01 (
   ACCT_NUM         INTEGER,
   CUST_LAST_NM     CHAR(15),
   LAST_ACTIVITY_DT VARCHAR(25),
   COL2             CHAR(10),
   COL3             CHAR(25),
   COL4             CHAR(25),
   COL5             CHAR(25),
   COL6             CHAR(55),
   STATE            CHAR(55))
 IN DBB.TS01

  PARTITION BY (ACCT_NUM)
   (PARTITION 1 ENDING AT (199),
    PARTITION 2 ENDING AT (299),
    PARTITION 3 ENDING AT (399),
    PARTITION 4 ENDING AT (MAXVALUE));
```

Example 11: Create a table, policy_info, that uses a SYSTEM_TIME period and create a history table, hist_policy_info. Then issue an ALTER TABLE statement to associate the policy_info table with the hist_policy_info table.

```
CREATE TABLE policy_info
    (policy_id CHAR(10) NOT NULL,
     coverage INT NOT NULL,
```

```
     sys_start TIMESTAMP(12) NOT NULL GENERATED ALWAYS AS ROW BEGIN,
     sys_end TIMESTAMP(12) NOT NULL GENERATED ALWAYS AS ROW END,
     create_id TIMESTAMP(12) GENERATED ALWAYS AS TRANSACTION START ID,
     PERIOD SYSTEM_TIME(sys_start,sys_end));
CREATE TABLE hist_policy_info
    (policy_id CHAR(10) NOT NULL,
     coverage INT NOT NULL,
     sys_start TIMESTAMP(12) NOT NULL,
     sys_end TIMESTAMP(12) NOT NULL,
     create_id TIMESTAMP(12));
ALTER TABLE policy_info
 ADD VERSIONING USE HISTORY TABLE hist_policy_info;
```

Example 12: Create a table, policy_info, that uses a BUSINESS_TIME period.

```
CREATE TABLE policy_info
 (policy_id CHAR(4) NOT NULL,
  coverage INT NOT NULL,
  bus_start DATE NOT NULL,
  bus_end DATE NOT NULL,
  PERIOD BUSINESS_TIME(bus_start, bus_end));
```

Example 13: Create a table, policy_info, that uses both a SYSTEM_TIME period and a BUSINESS_TIME period to keep historical rows and track a user-specified time period. A table that specifies both a SYSTEM_TIME period and a BUSINESS_TIME period is sometimes referred to as a *bitemporal table*. To enable retention of historical rows, a history table, hist_policy_info, also needs to be created and associated (using the ALTER TABLE statement) with the policy_info table.

```
CREATE TABLE policy_info
 (policy_id CHAR(4) NOT NULL,
 coverage INT NOT NULL,
 bus_start DATE NOT NULL,
 bus_end DATE NOT NULL,
 sys_start TIMESTAMP(12) NOT NULL GENERATED ALWAYS AS ROW BEGIN,
 sys_end TIMESTAMP(12) NOT NULL GENERATED ALWAYS AS ROW END,
 create_id TIMESTAMP(12) GENERATED ALWAYS AS TRANSACTION START ID,
 PERIOD BUSINESS_TIME(bus_start, bus_end),
 PERIOD SYSTEM_TIME(sys_start, sys_end));
CREATE TABLE hist_policy_info
 (policy_id CHAR(4) NOT NULL,
 coverage INT NOT NULL,
 bus_start DATE NOT NULL,
 bus_end DATE NOT NULL,
 sys_start TIMESTAMP(12) NOT NULL,
 sys_end TIMESTAMP(12) NOT NULL),
 create_id TIMESTAMP(12);
ALTER TABLE policy_info
 ADD VERSIONING USE HISTORY TABLE hist_policy_info;
```

CREATE TABLESPACE

The CREATE TABLESPACE statement defines a segmented, partitioned, or universal table space at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set that is defined below must include at least one of the following:

- The CREATETS privilege for the database
- DBADM, DBCTRL, or DBMAINT authority for the database
- SYSADM or SYSCTRL authority
- System DBADM

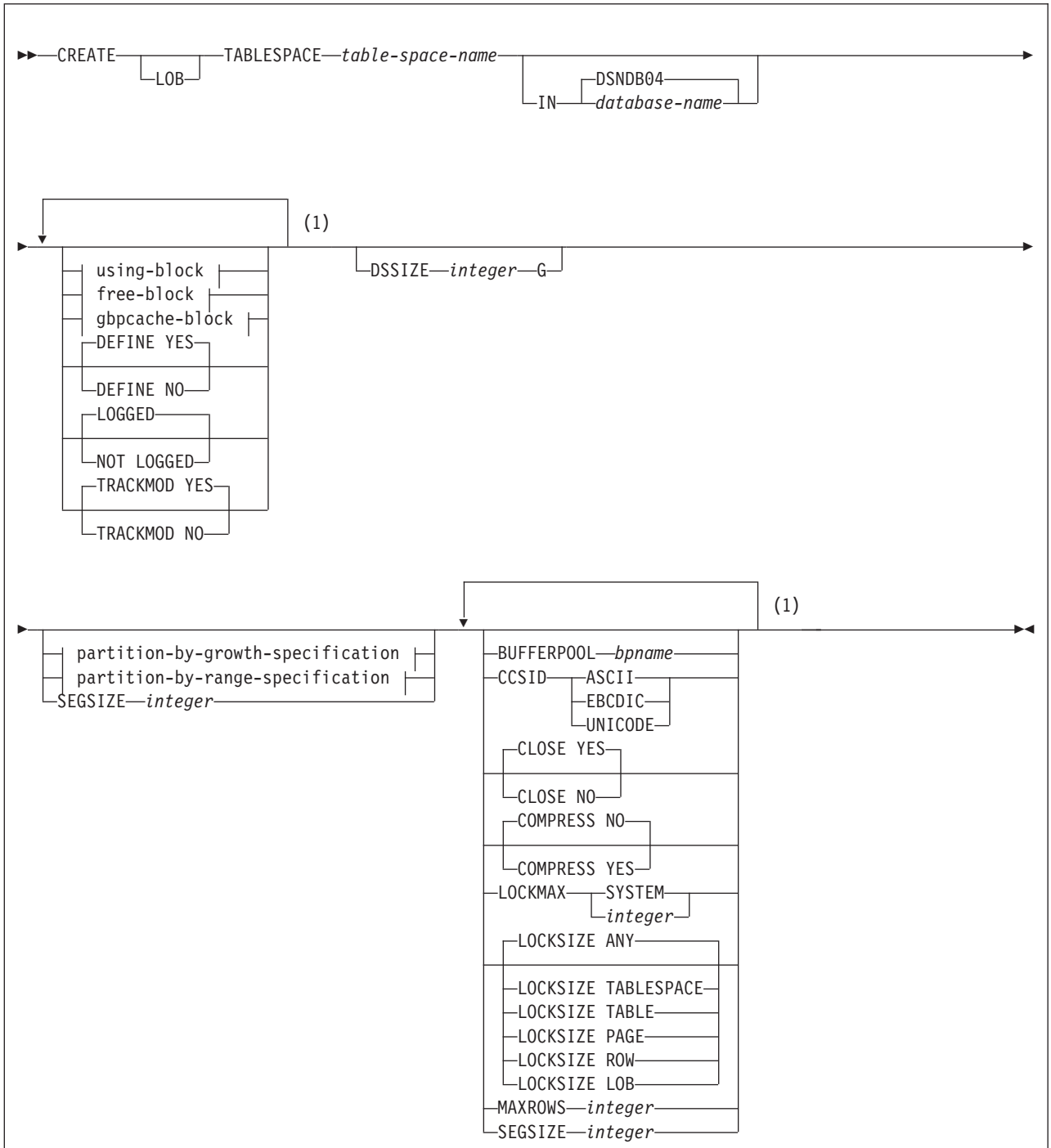
If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

Additional privileges might be required, as explained in the description of the BUFFERPOOL and USING STOGROUP clauses.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the of the owner of the plan or package. If the application is bound in a trusted context with the ROLE AS OBJECT OWNER clause specified, a role is the owner. Otherwise, an authorization ID is the owner.

If the statement is dynamically prepared, the privilege set is the privileges that are held by the SQL authorization ID of the process unless the process is within a trusted context and the ROLE AS OBJECT OWNER clause is specified. In that case, the privileges set is the privileges that are held by the role that is associated with the primary authorization ID of the process.

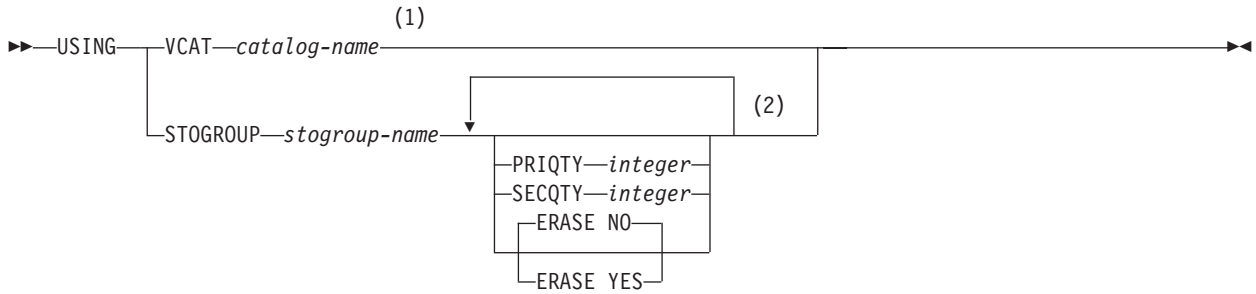
Syntax



Notes:

1 The same clause must not be specified more than one time.

using-block:

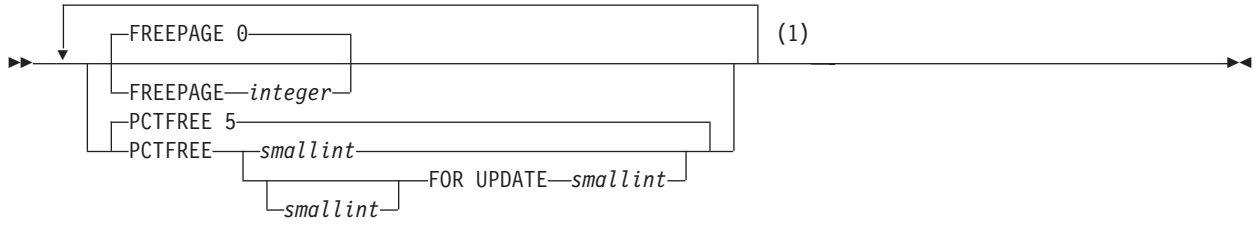


Notes:

1 USING VCAT must not be specified if MAXPARTITIONS is also specified.

2 The same clause must not be specified more than one time.

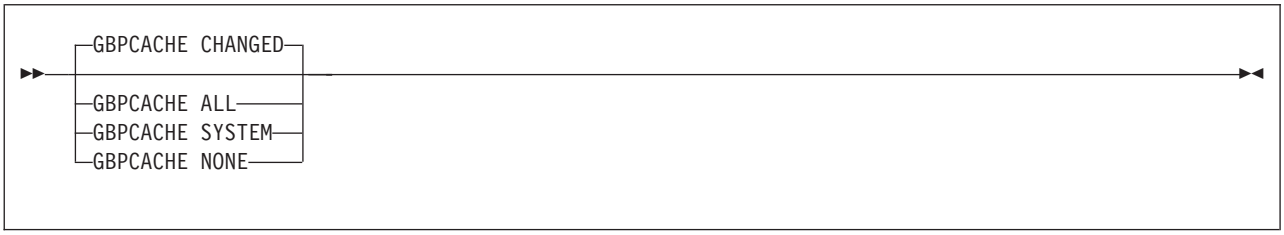
free-block:



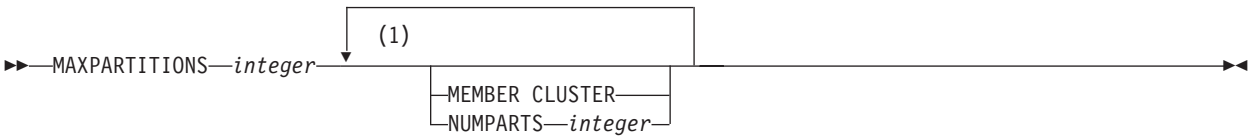
Notes:

1 The same clause must not be specified more than one time.

gbpcache-block:



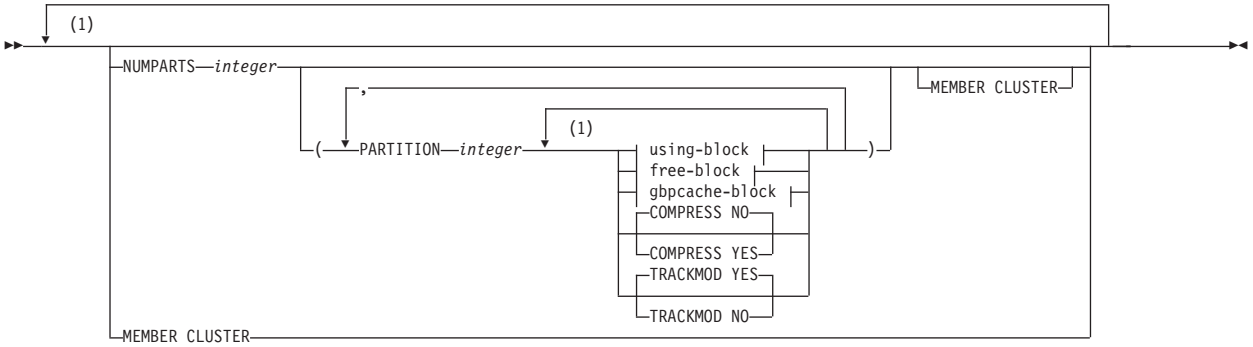
partition-by-growth-specification:



Notes:

1 The same clause must not be specified more than one time

partition-by-range-specification:



Notes:

1 The same clause must not be specified more than one time

Description

LOB

Identifies the table space as LOB table space. A LOB table space is used to hold LOB values.

The LOB table space must be in the same database as its associated base table space. Do not specify LOB for a table space in a work file database.

table-space-name

Names the table space. The name, qualified with the *database-name* implicitly or explicitly specified by the IN clause, must not identify a table space, index space, or LOB table space that exists at the current serveror that exists in the SYSPENDINGOBJECTS catalog table.

A table space that is for declared temporary tables must be in the work file database. PUBLIC implicitly receives the USE privilege (without GRANT authority) on any table space created in the work file database. This implicit privilege is not recorded in the DB2 catalog, and it cannot be revoked.

IN *database-name*

Specifies the database in which the table space is created. *database-name* must identify a database that exists at the current server and must not specify the following:

- DSNDB06 for any type of table space
- A work file database for a LOB table space

- A TEMP database
- An implicitly created database

If the table space is for declared temporary tables or static scrollable cursors, the name of the work file database must be specified.

DSNDB04 is the default.

The components of the USING clause are discussed below, first for nonpartitioned table spaces and then for partitioned table spaces. If you omit USING, the default storage group of the database must exist.

USING clause for nonpartitioned table spaces:

For nonpartitioned table spaces, the USING clause indicates whether the data set for the table space is defined by you or by DB2. If DB2 is to define the data set, the clause also gives space allocation parameters and an erase rule.

If you omit USING, DB2 defines the data sets using the default storage group of the database and the defaults for PRIQTY, SECQTY, and ERASE.

VCAT *catalog-name*

Specifies that the first data set for the table space is managed by the user, and following data sets, if needed, are also managed by the user.

The data sets defined for the table space are linear VSAM data sets cataloged in an integrated catalog facility catalog identified by *catalog-name*. An alias[33] must be used if the catalog name is longer than eight characters.

Conventions for table space data set names are given in *DB2 Administration Guide*. *catalog-name* is the first qualifier for each data set name.

One or more DB2 subsystems could share integrated catalog facility catalogs with the current server. To avoid the chance of having one of those subsystems attempt to assign the same name to different data sets, select a value for *catalog-name* that is not used by the other DB2 subsystems.

VCAT must not be specified if MAXPARTITIONS is also specified.

STOGROUP *stogroup-name*

Specifies that DB2 will define and manage the data sets for the table space. Each data set will be defined on a volume of the identified storage group. The values specified (or the defaults) for PRIQTY and SECQTY determine the primary and secondary allocations for the data set. The storage group supplies the name of a volume for the data set and the first-level qualifier for the data set name. The first-level qualifier is also the name of, or an alias[33] for, the integrated catalog facility catalog on which the data set is to be cataloged. The naming conventions for the data set are the same as if the data set is managed by the user. As was mentioned above for VCAT, the first-level qualifier could cause naming conflicts if the local DB2 can share integrated catalog facility catalogs with other DB2 subsystems.

stogroup-name must identify a storage group that exists at the current server. SYSADM or SYSCTRL authority, or the USE privilege on the storage group, is required.

The description of the storage group must include at least one volume serial number, or it must indicate that the choice of volumes is left to Storage Management Subsystem (SMS). If volume serial numbers appear in

33. The alias of an integrated catalog facility catalog.

the description, each must identify a volume that is accessible to z/OS for dynamic allocation of the data set, and all identified volumes must be of the same device type.

The integrated catalog facility catalog used for the storage group must ***not*** contain an entry for the first data set of the table space. If the integrated catalog facility catalog is password protected, the description of the storage group must include a valid password.

PRIQTY *integer*

Specifies the minimum primary space allocation for a DB2-managed data set. When you specify PRIQTY with a value other than -1, the primary space allocation is at least *n* kilobytes, where *n* is the value of *integer* with the following exceptions:

- For 4KB page sizes, if *integer* is less than 12, *n* is 12.
- For 8KB page sizes, if *integer* is less than 24, *n* is 24.
- For 16KB page sizes, if *integer* is less than 48, *n* is 48.
- For 32KB page sizes, if *integer* is less than 96, *n* is 96.
- For any page size, if *integer* is greater than 67108864, *n* is 67108864.

For LOB table spaces, the exceptions are:

- For 4KB page sizes, if *integer* is less than 200, *n* is 200.
- For 8KB page sizes, if *integer* is less than 400, *n* is 400.
- For 16KB page sizes, if *integer* is less than 800, *n* is 800.
- For 32KB page sizes, if *integer* is less than 1600, *n* is 1600.
- For any page size, if *integer* is greater than 67108864, *n* is 67108864.

If you do not specify PRIQTY or specify PRIQTY -1, DB2 uses a default value for the primary space allocation; for information on how DB2 determines the default value, see Rules for primary and secondary space allocation.

If you specify PRIQTY and do not specify a value of -1, DB2 specifies the primary space allocation to access method services using the smallest multiple of *p* KB not less than *n*, where *p* is the page size of the table space. The allocated space can be greater than the amount of space requested by DB2. For example, it could be the smallest number of tracks that will accommodate the request. The amount of storage space requested must be available on some volume in the storage group based on VSAM space allocation restrictions. Otherwise, the primary space allocation will fail. To more closely estimate the actual amount of storage, see DEFINE CLUSTER command (DFSMS Access Method Services for Catalogs).

Executing this statement causes only one data set to be created. However, you might have more data than this one data set can hold. DB2 automatically defines more data sets when they are needed. Regardless of the value in PRIQTY, when a data set reaches its maximum size, DB2 creates a new one. To enable a data set to reach its maximum size without running out of extents, it is recommended that you allow DB2 to automatically choose the value of the secondary space allocations for extents.

If you do choose to explicitly specify SECQTY, to avoid wasting space, use the following formula to make sure that PRIQTY and its associated secondary extent values do not exceed the maximum size of the data set:

```
PRIQTY + (number of extents * SECQTY) <= DSSIZE (implicit or explicit)
```

SECQTY *integer*
Specifies the minimum secondary space allocation for a DB2-managed data set. If you do not specify SECQTY, DB2 uses a formula to determine a value. For information on the actual value that is used for secondary space allocation, whether you specify a value or not, see Rules for primary and secondary space allocation.

If you specify SECQTY and do not specify a value of -1, DB2 specifies the secondary space allocation to access method services using the smallest multiple of *p* KB not less than *n*, where *p* is the page size of the table space. The allocated space can be greater than the amount of space requested by DB2. For example, it could be the smallest number of tracks that will accommodate the request. To more closely estimate the actual amount of storage, see DEFINE CLUSTER command (DFSMS Access Method Services for Catalogs).

ERASE
Indicates whether the DB2-managed data sets for the table space or partition are to be erased when they are deleted during the execution of a utility or an SQL statement that drops the table space.

NO Does not erase the data sets. Operations involving data set deletion will perform better than ERASE YES. However, the data is still accessible, though not through DB2. This is the default.

YES
Erases the data sets. As a security measure, DB2 overwrites all data in the data sets with zeros before they are deleted.

USING clause for partitioned table spaces:
If the table space is partitioned, there is a USING clause for each partition; either one you give explicitly or one provided by default. Except as explained below, the meaning of the clause and the rules that apply to it are the same as for a nonpartitioned table space.

The USING clause for a particular partition is the first of these choices that can be found:

- A USING clause in the PARTITION clause for the partition
- A USING clause that is not in any PARTITION clause
- An implicit USING STOGROUP clause that identifies the default storage group of the database and accepts the defaults for PRIQTY, SECQTY, and ERASE

VCAT *catalog-name*
Indicates that the data set for the partition is managed by the user using the naming conventions set forth in *DB2 Administration Guide*. As was true for the nonpartitioned case, *catalog-name* identifies the catalog for the data set and supplies the first-level qualifier for the data set name.

One or more DB2 subsystems could share integrated catalog facility catalogs with the current server. To avoid the chance of having one of those subsystems attempt to assign the same name to different data sets, select a value for *catalog-name* that is not used by the other DB2 subsystems.

DB2 assumes one and only one data set for each partition.

STOGROUP *stogroup-name*
Indicates that DB2 will create a data set for the partition with the aid of a

storage group named *stogroup-name*. The data set is defined during the execution of this statement. DB2 assumes one and only one data set for each partition.

The *stogroup-name* must identify a storage group that exists at the current server and the privilege set must include SYSADM authority, SYSCTRL authority, or the USE privilege for the storage group. The integrated catalog facility catalog used for the storage group must ***not*** contain an entry for that data set.

When USING STOGROUP is specified for a partition, the defaults for PRIQTY, SECQTY, and ERASE are the values specified in the USING STOGROUP clause that is not in any PARTITION clause. If that USING STOGROUP clause is not specified, the defaults are those specified in the description of PRIQTY, SECQTY, and ERASE.

FREEPAGE *integer*

Specifies how often to leave a page of free space when the table space or partition is loaded or reorganized. You must specify an integer in the range 0 to 255. If you specify 0, no pages are left as free space. Otherwise, one free page is left after every *n* pages, where *n* is the specified *integer* value. However, if the table space is segmented and the integer you specify is not less than the segment size, *n* is one less than the segment size.

If the table space is segmented, the number of pages left free must be less than the SEGSIZE value. If the number of pages to be left free is greater than or equal to the SEGSIZE value, then the number of pages is adjusted downward to one less than the SEGSIZE value.

The default is FREEPAGE 0, leaving no free pages. Do not specify FREEPAGE for a LOB table space or a table space in a work file database.

For XML table spaces, this change has no effect until data in the table space is reorganized.

FREEPAGE does not apply to hash-organized table spaces.

Related information:

Reserving free space for table spaces (DB2 Performance)

Reserving free spaces for indexes (DB2 Performance)

PCTFREE *smallint*

Indicates what percentage of each page to leave as free space when the table is loaded or reorganized. *smallint* in the range from 0 to 99. The first record on each page is loaded without restriction. When additional records are loaded, at least *smallint* percent of free space is left on each page.

The default is PCTFREE 5, which means that 5% of the space on each pages is reserved as free space. Do not specify PCTFREE for a LOB table space or a table space in a work file database.

For XML table spaces, this change has no effect until data in the table space is reorganized.

PCTFREE does not apply to table spaces for hash-organized tables except when AUTOESTSPACE(YES) is specified in a REORG TABLESPACE invocation.

FOR UPDATE *smallint*

Specifies the percentage of space to reserve as free space on each page, for use by subsequent UPDATE operations, when data is added to the table by

INSERT operations or utilities. The *smallint* value is an integer in the range -1 to 99. FOR UPDATE -1 specifies that 5% of free space is reserved initially, and the amount of free spaces is calculated automatically based on certain real-time statistics values. The first record on each page is loaded always loaded without restriction.

If this value is not specified, the value of the PCTFREE_UPD subsystem parameter is used.

The value is recorded in the PCTFREE_UPD column of the SYSIBM.SYSTABLEPART catalog table.

The FOR UPDATE *smallint* values do not apply to LOB table spaces, XML table spaces, or table spaces that use hash organization.

The sum of the values for PCTFREE *smallint* and FOR UPDATE*smallint* must be less than or equal to 99.

If the table space is partitioned, the values of FREEPAGE and PCTFREE for a particular partition are given by the first of these choices that apply:

- The values of FREEPAGE and PCTFREE given in the PARTITION clause for that partition
- The values given in a *free-block* that is not in any PARTITION clause
- The default values are FREEPAGE 0 and PCTFREE 5.

Related information:

Reserving free space for table spaces (DB2 Performance)

Reserving free spaces for indexes (DB2 Performance)

GBPCACHE

In a data sharing environment, specifies what pages of the table space or partition are written to the group buffer pool in a data sharing environment. In a non-data-sharing environment, you can specify GBPCACHE for a table space other than one in a work file database, but it is ignored. Do not specify GBPCACHE for a table space in a work file database in either environment (data sharing or non-data-sharing).

CHANGED

When there is inter-DB2 R/W interest on the table space or partition, updated pages are written to the group buffer pool. When there is no inter-DB2 R/W interest, the group buffer pool is not used. Inter-DB2 R/W interest exists when more than one member in the data sharing group has the table space or partition open, and at least one member has it open for update. GBPCACHE CHANGED is the default.

If the table space is in a group buffer pool that is defined to be used only for cross-invalidation (GBPCACHE NO), CHANGED is ignored and no pages are cached to the group buffer pool.

ALL

Indicates that pages are to be cached in the group buffer pool as they are read in from DASD.

Exception: In the case of a single updating DB2 when no other DB2s have any interest in the page set, no pages are cached in the group buffer pool.

If the table space is in a group buffer pool that is defined to be used only for cross-invalidation (GBPCACHE NO), ALL is ignored and no pages are cached to the group buffer pool.

SYSTEM
Indicates that only changed system pages within the LOB table space are to be cached to the group buffer pool. A system page is a space map page or any other page that does not contain actual data values.

Use SYSTEM only for a LOB table space.

NONE
Indicates that no pages are to be cached to the group buffer pool. DB2 uses the group buffer pool only for cross-invalidation.

If you specify NONE, the table space or partition must not be in recover pending status and must be in the stopped state when the CREATE TABLESPACE statement is executed.

If the table space is partitioned, the value of GBPCACHE for a particular partition is given by the first of these choices that applies:

1. The value of GBPCACHE given in the PARTITION clause for that partition. Do not use more than one *gbpcache-block* in any PARTITION clause.
2. The value given in a *gbpcache-block* that is not in any PARTITION clause.
3. The default value CHANGED.

DEFINE
Specifies when the underlying data sets for the table space are physically created.

YES
The data sets are created when the table space is created (the CREATE TABLESPACE statement is executed). YES is the default.

If MAXPARTITIONS is also specified, only the first partition is created when the table space is created. Additional partitions are created as needed.

NO The data sets are not created until data is inserted into the table space. DEFINE NO is applicable only for DB2-managed data sets (USING STOGROUP is specified). DEFINE NO is ignored for user-managed data sets (USING VCAT is specified). DB2 uses the SPACE column in catalog table SYSTABLEPART to record the status of the data sets (undefined or allocated).

Do not specify DEFINE NO for a table space in a work file database. DEFINE NO is not recommended if you intend to use any tools outside of DB2 to manipulate data, such as to load data, because data sets might then exist when DB2 does not expect them to exist. When DB2 encounters this inconsistent state, applications will receive an error.

For table spaces that are created with DEFINE NO, point-in-time recover will not work before data sets exist and before a recovery copy exists.

LOGGED or NOT LOGGED
Specifies whether changes that are made to the data in the specified table space are recorded in the log.

LOGGED
Specifies that changes that are made to the data in the specified table space are recorded in the log. This applies to all tables that are created in the specified table space and to all indexes of those tables. XML table spaces and their indexes inherit the logging attribute from the associated base table space. Auxiliary indexes also inherit the logging attribute from the associated base table space.

LOGGED cannot be specified for table spaces in DSNDB06 (the DB2 catalog) or in a work file database.

LOGGED is the default.

NOT LOGGED
Specifies that changes that are made to data in the specified table space are not recorded in the log. This parameter applies to all tables that are created in the specified table space and to all indexes of those tables. XML table spaces and their indexes inherit the logging attribute from the associated base table space. Auxiliary indexes inherit the logging attribute from the associated base table space.

NOT LOGGED prevents undo and redo information from being recorded in the log; however, control information for the specified table space will continue to be recorded in the log.

NOT LOGGED cannot be specified for table spaces in the following databases:

- DSNDB06 (the DB2 catalog)
- a work file database

TRACKMOD
Specifies whether DB2 tracks modified pages in the space map pages of the table space or partition. Do not specify TRACKMOD for a LOB table space. Do not specify TRACKMOD for a table space in a work file database.

YES
DB2 tracks changed pages in the space map pages to improve the performance of incremental image copy.

NO DB2 does not track changed pages in the space map pages. It uses the LRSN value in each page to determine whether a page has been changed.

If the table space is partitioned, the value of TRACKMOD for a particular partition is given by the first of these choices that applies:

1. The value of TRACKMOD given in the PARTITION clause for that partition.
2. The value given in a *trackmod-block* that is not in any PARTITION clause.
3. The default value YES.

If TRACKMOD is not specified, the default value as specified in the subsystem parameter TRACK MODIFIED PAGES is used.

DSSIZE *integer***G**
Specifies the maximum size for each partition, or for LOB table spaces, each data set. If you specify DSSIZE, you must also specify the NUMPARTS, MAXPARTITIONS, or LOB clause. Do not specify DSSIZE for a table space in a work file database unless MAXPARTITIONS is also specified.

The following values are valid:
1G 1 gigabyte
2G 2 gigabytes
4G 4 gigabytes
8G 8 gigabytes
16G 16 gigabytes
32G 32 gigabytes
64G 64 gigabytes
128G 128 gigabytes
256G 256 gigabytes

To specify a value greater than 4G, the data sets for the table space must be associated with a DFSMS data class that has been specified with extended format and extended addressability.

If NUMPARTS is also specified, the maximum size of each partition depends on the value of NUMPARTS, as shown in the following table. Otherwise, the maximum size of each partition is 4G.

Table 122. Maximum partition size depending on value of NUMPARTS

Value of NUMPARTS	Maximum partition size (default for DSSIZE)
1 to 16	4GB (4G)
17 to 32	2GB (2G)
33 to 64	1GB (1G)
65 to 254	4GB (4G)

If NUMPARTS is greater than 254, the maximum partition size (and the default for DSSIZE) depends on the page size of the table space, as shown in the following table. The partition size shown is not necessarily the actual number of bytes used or allocated for any one partition; it is the largest number that can be logically addressed. Each partition occupies one data set.

Table 123. Maximum partition size depending on page size

Page size	Maximum partition size (default for DSSIZE)
4K	4GB (4G)
8K	8GB (8G)
16K	16GB (16G)
32K	32GB (32G)

If DSSIZE is explicitly specified, the maximum number of partitions that can be specified or is the default is limited by the maximum table space size. For example:

- For a partitioned table space with a 4K page size, if DSSIZE 64GB is specified, the maximum NUMPARTS value is 256.
- For a partitioned table space with an 8K page size, if DSSIZE 64GB is specified, the maximum NUMPARTS value is 512.
- For a partitioned table space with a 32K page size, if DSSIZE 128GB is specified, the maximum NUMPARTS value is 1024.

See Table 125 on page 1468 for more information on the relationship between DSSIZE, NUMPARTS, and the table space size.

For LOB table spaces, if DSSIZE is not specified, the default for the maximum size of each data set is 4GB. The maximum number of data sets is 254.

For a description of the maximum size of a LOB table space (or the maximum size of LOB data for each column of a base table), see Large object table spaces (Introduction to DB2 for z/OS).

MAXPARTITIONS *integer*

Specifies that the table space is a partition-by-growth table space. The data set for the first partition is allocated unless the DEFINE NO clause is specified for the partition. The data sets for additional partitions are not allocated until they are needed.

integer specifies the maximum number of partitions to which the table space can grow. *integer* must be in the range of 1 to 4096, depending on the corresponding value of the DSSIZE clause. The following table shows the maximum value for MAXPARTITIONS in relation to the page size or DSSIZE value for the table space.

Table 124. Maximum value for MAXPARTITIONS given the page size or DSSIZE value for the table space

DSSIZE value	4K page size	8K page size	16K page size	32K page size
1G - 4G (1 GB to 4 GB)	4096	4096	4096	4096
8G (8 GB)	2048	4096	4096	4096
16G (16 GB)	1024	2048	4096	4096
32G (32 GB)	512	1024	2048	4096
64G (64 GB)	254	512	1024	2048
128G (128 GB)	128	256	512	1024
256G (256 GB)	64	128	256	512

If MAXPARTITIONS is specified for a table space in a work file database, the table space will be a partition-by-growth table space. If the DSSIZE or NUMPARTS clauses are not specified in addition to the MAXPARTITIONS clause for a table space in a work file database, the default values for DSSIZE and NUMPARTS will be used. If MAXPARTITIONS is not specified for a table space in a work file database, the table space will be a segmented table space.

Although physical data sets are not defined when the MAXPARTITIONS value is issued, there can be storage and cpu overhead. If an increase in the number of partitions is expected by using the MAXPARTITONS clause, be aware that specifying an value larger than necessary, such as 4096 (the maximum value), as a default for all of your partition-by-growth table spaces can cause larger than expected storage requests.

MEMBER CLUSTER

Specifies that data inserted by an insert operation is not clustered by the implicit clustering index (the first index) or the explicit clustering index. Instead, DB2 chooses where to locate the data in the table space based on available space.

Do not specify MEMBER CLUSTER for segmented table spaces.

Do not specify MEMBER CLUSTER for a LOB table space or a table space in a work file database.

MEMBER CLUSTER can be specified with MAXPARTITIONS.

NUMPARTS *integer*

Indicates that the table space is partitioned. If MAXPARTITIONS is also specified, the table space is a partition-by-growth table space, otherwise, the table space is a range-partitioned table space.

integer

Specifies the number of partitions. If *n* is specified for a partition-by-growth table space, *integer* indicates the number of partitions that are initially created unless DEFINE NO is specified. *integer* must be a value between 1 and 4096 inclusive and must be less than or equal to the value that is specified for the MAXPARTITIONS clause.

The maximum size of each partition depends on the value that is specified for DSSIZE. If DSSIZE is not specified, the number of partitions that are specified determines the maximum size of each partition. For a summary of the values for the maximum size of each partition, see Table 122 on page 1466 and Table 123 on page 1466.

The maximum number of partitions that a table space can have depends on the page size and DSSIZE. The total table space size depends on how many partitions it has and DSSIZE. Page size affects table size because it affects how many partitions a table space can have. The following table shows the maximum number of partitions for DSSIZE and page size and total table space size for both EA-enabled (extended addressability) and non-EA-enabled data sets. (Specifying a DSSIZE greater than 4GB requires EA-enabled data sets.)

Table 125. Table space size given page size and partitions

Type of RID	Page size	DSSIZE	Maximum number of partitions	Total table space size
5-byte EA	4KB	1GB	4096	4TB
5-byte EA	4KB	2GB	4096	8TB
5-byte EA	4KB	4GB	4096	16TB
5-byte EA	4KB	8GB	2048	16TB
5-byte EA	4KB	16GB	1024	16TB
5-byte EA	4KB	32GB	512	16TB
5-byte EA	4KB	64GB	256	16TB
5-byte EA	4KB	128GB	128	16TB
5-byte EA	4KB	256GB	64	16TB
5-byte EA	8KB	1GB	4096	4TB
5-byte EA	8KB	2GB	4096	8TB
5-byte EA	8KB	4GB	4096	16TB
5-byte EA	8KB	8GB	4096	32TB
5-byte EA	8KB	16GB	2048	32TB
5-byte EA	8KB	32GB	1024	32TB
5-byte EA	8KB	64GB	512	32TB
5-byte EA	8KB	128GB	256	32TB
5-byte EA	8KB	256GB	128	32TB
5-byte EA	16KB	1GB	4096	4TB
5-byte EA	16KB	2GB	4096	8TB
5-byte EA	16KB	4GB	4096	16TB
5-byte EA	16KB	8GB	4096	32TB
5-byte EA	16KB	16GB	4096	64TB
5-byte EA	16KB	32GB	2048	64TB
5-byte EA	16KB	64GB	1024	64TB
5-byte EA	16KB	128GB	512	64TB
5-byte EA	16KB	256GB	256	64TB
5-byte EA	32KB	1GB	4096	4TB
5-byte EA	32KB	2GB	4096	8TB

Table 125. Table space size given page size and partitions (continued)

Type of RID	Page size	DSSIZE	Maximum number of partitions	Total table space size
5-byte EA	32KB	4GB	4096	16TB
5-byte EA	32KB	8GB	4096	32TB
5-byte EA	32KB	16GB	4096	64TB
5-byte EA	32KB	32GB	4096	128TB
5-byte EA	32KB	64GB	2048	128TB
5-byte EA	32KB	128GB	1024	128TB
5-byte EA	32KB	256GB	512	128TB
5-byte (non-EA) LARGE	4KB	(4GB)	4096	16TB

For a description of the maximum size of a LOB table space (or the maximum size of LOB data for each column of a base table), see Large object table spaces (Introduction to DB2 for z/OS).

If you omit NUMPARTS, the table space is segmented with a SEGSIZE of 4, LOCKSIZE ANY (unless it is explicitly specified), is not partitioned, and initially occupies one data set.

Do not specify NUMPARTS for a LOB table space. Do not specify NUMPARTS for a table space in a work file database unless the MAXPARTITIONS clause is also specified.

PARTITION *integer*
Specifies to which partition the following *using-block* or *free-block* applies. *integer* can range from 1 to the number of partitions given by NUMPARTS.

You can code the PARTITION clause (and any *using-block* or *free-block* that follows it) as many times as needed. If you use the same partition number more than once, only the last specification for that partition is used.

The PARTITION clause must not be specified if the table space is a partition-by-growth table space.

BUFFERPOOL *bpname*
Identifies the buffer pool to be used for the table space and determines the page size of the table space. For 4KB, 8KB, 16KB and 32KB page buffer pools, the page sizes are 4 KB, 8 KB, 16 KB, and 32 KB, respectively. The *bpname* must identify an activated buffer pool, and the privilege set must include SYSADM or SYSCTRL authority, or the USE privilege on the buffer pool. If the table space is to be created in a work file database, you cannot specify 8KB and 16KB buffer pools.

If you do not specify the BUFFERPOOL clause, the default buffer pool of the database is used unless the table space that is being created is a LOB table space. If you do not specify the BUFFERPOOL clause and the table space that is being created is a LOB table space, the default buffer pool is the buffer pool that is specified in the DEFAULT BUFFER POOL FOR USER LOB DATA field on installation panel DSNTIP1.

See “Naming conventions” on page 57 for more details about *bpname*. See -ALTER BUFFERPOOL (DB2) (DB2 Commands) for a description of active and inactive buffer pools.

LOCKSIZE
Specifies the size of locks used within the table space and, in some cases, also the threshold at which lock escalation occurs. Do not use this clause for a table space in a work file database.

ANY
Specifies that DB2 can use any lock size.

In most cases, DB2 uses LOCKSIZE PAGE LOCKMAX SYSTEM for non-LOB table spaces and LOCKSIZE LOB LOCKMAX SYSTEM for LOB table spaces. However, when the number of locks acquired for the table space exceeds the maximum number of locks allowed for a table space (the NUMLKTS subsystem parameter), the page or LOB locks are released and locking is set at the next higher level.

If the table space is segmented, the next higher level is the table. If the table space is segmented and not partitioned, the next higher level is the table. If the table space is partitioned, the next higher level is the partition.

If the table space is implicitly created, DB2 uses LOCKSIZE ROW.

TABLESPACE
Specifies table space locks.

TABLE
Specifies table locks. Use TABLE only for a segmented table space. Do not use TABLE for a universal table space.

PAGE
Specifies page locks. Do not use PAGE for a LOB table space.

ROW
Specifies row locks. Do not use ROW for a LOB table space.

LOB
Specifies LOB locks. Use LOB only for a LOB table space.

LOCKMAX
Specifies the maximum number of page, row, or LOB locks an application process can hold simultaneously in the table space. If a program requests more than that number, locks are escalated. The page, row, or LOB locks are released and the intent lock on the table space or segmented table is promoted to S or X mode. If you specify LOCKMAX for a table space in a work file database, DB2 ignores the value because these types of locks are not used.

integer
Specifies the number of locks allowed before escalating, in the range 0 to 2 147 483 647.

Zero (0) indicates that the number of locks on the table or table space are not counted and escalation does not occur.

SYSTEM
Indicates that the value of LOCKS PER TABLE(SPACE), on installation panel DSNTIPJ, specifies the maximum number of page, row, or LOB locks a program can hold simultaneously in the table or table space.

The following table summarizes the results of specifying a LOCKSIZE value while omitting LOCKMAX.

LOCKSIZE	Resultant LOCKMAX
ANY	SYSTEM

LOCKSIZE	Resultant LOCKMAX
TABLESPACE, TABLE, PAGE, ROW, or LOB	0

If the lock size is TABLESPACE or TABLE, LOCKMAX must be omitted, or its operand must be 0.

CLOSE
When the limit on the number of open data sets is reached, specifies the priority in which data sets are closed.

YES
Eligible for closing before CLOSE NO data sets. This is the default unless the table space is in a work file database.

NO Eligible for closing after all eligible CLOSE YES data sets are closed.

For a table space in a work file database, DB2 uses CLOSE NO regardless of the value specified.

COMPRESS
Specifies whether data compression applies to the rows of the table space or partition. Do not specify COMPRESS for a LOB table space or a table space in a work file database.

For partitioned table spaces, the COMPRESS attribute for each partition is the value from the first of the following conditions that apply:

- The value specified in the COMPRESS clause in the PARTITION clause for the partition
- The value specified in the COMPRESS clause that is not in any PARTITION clause
- An implicit COMPRESS NO by default.

For more information about data compression, see Compressing your data (DB2 Performance).

YES
Specifies data compression. The rows are not compressed until the LOAD or REORG utility is run on the table in the table space or partition, or until an insert operation is performed through the INSERT statement or the MERGE statement.

NO Specifies no data compression for the table space or partition.

CCSID *encoding-scheme*
Specifies the encoding scheme for tables stored in the table space.

If you do not specify a CCSID when it is allowed, the default is the encoding scheme of the database in which the table space resides, except for table spaces in database DSNDB04; for table spaces in DSNDB04, the default is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

ASCII Specifies that the data is to be encoded using ASCII CCSIDs. If the database in which the table space is to reside is already defined as ASCII, the ASCII CCSIDs associated with that database are used. Otherwise, the default ASCII CCSIDs of the server are used.

EBCDIC
Specifies that the data is to be encoded using EBCDIC CCSIDs.

UNICODE
Specifies that the data is to be encoded using the UNICODE CCSIDs of the server.

Usually, each encoding scheme requires only a single CCSID. Additional CCSIDs are needed when mixed, graphic, or Unicode data is used.

All data stored within a table space must use the same encoding scheme unless the table space is in a work file database.

Do not specify CCSID for a LOB table space or a table space in a work file database. The encoding scheme for a LOB table space is inherited from the base table space. A table space in a work file database does not have an associated encoding scheme because the table space can contain created and declared temporary tables with a mixture of encoding schemes.

MAXROWS *integer*
Specifies the maximum number of rows that DB2 will consider placing on each data page. The integer can range from 1 through 255. This value is considered for insert operations, LOAD, and REORG. For LOAD and REORG (which do not apply for a table space in the work file database), the PCTFREE specification is considered before MAXROWS; therefore, fewer rows might be stored than the value you specify for MAXROWS.

If you do not specify MAXROWS, the default number of rows is 255.

Do not use MAXROWS for a LOB table space or a table space in a work file database.

SEGSIZE *integer*
Specifies the type of table space that will be created depending on the values of the SEGSIZE, MAXPARTITIONS, and NUMPARTS clauses. *integer* specifies the number of pages that are to be assigned to each segment of the table space. *integer* must be a multiple of 4 between 0 and 64 (inclusive). *integer* cannot be 0 for an XML table space. Do not specify SEGSIZE for a LOB table space.

If SEGSIZE is not specified and only the NUMPARTS clause is specified, either a partitioned table space or a range-partitioned universal table space is created depending on the value of the DPSEGSZ subsystem parameter:

- If the value of DPSEGSZ is greater than 0 (zero), the table space will be a range-partitioned universal table with a SEGSIZE value that is equal to the value of DPSEGSZ.
- If the value of DPSEGSZ is equal to 0 (zero), the table space will be a partitioned table space.

If SEGSIZE is not specified and only the MAXPARTITIONS clause is specified, a partition-by-growth universal table space is created with a SEGSIZE that depends on the value of the DPSEGSZ subsystem parameter:

- If the value of DPSEGSZ is greater than 0 (zero), the table space will have a SEGSIZE value that is equal to the value of DPSEGSZ.
- If the value of DPSEGSZ is equal to 0 (zero), the table space will have a SEGSIZE value of 32.

The DPSEGSZ subsystem parameter has no effect on segmented table spaces.

The following table lists the type of table space depending on the specification of the SEGSIZE, MAXPARTITIONS, and NUMPARTS clauses:

Table 126. Type of table space depending on value of SEGSIZE, MAXPARTITIONS, and NUMPARTS clauses

SEGSIZE clause	MAXPARTITIONS clause	NUMPARTS clause	Type of table space
specified	specified	not specified	partition-by-growth table space
specified	not specified	not specified	segmented table space
specified	not specified	specified	range-partitioned universal table space
specified with a value of 0	not specified	specified	partitioned table space
not specified	specified	specified	partition-by-growth table space • SEGSIZE = 32 if DPSEGSZ = 0 • SEGSIZE = *n* if DPSEGSZ = *n*, where *n* is a non-zero value for DPSEGSZ.
not specified	specified	not specified	partition-by-growth table space • SEGSIZE = 32 if DPSEGSZ = 0 • SEGSIZE = *n* if DPSEGSZ = *n*, where *n* is a non-zero value for DPSEGSZ.
not specified	not specified	specified	One of the following: • partitioned table space if DPSEGSZ is specified with a value of 0. • range-partitioned universal table space with a SEGSIZE = *n* if DPSEGSZ = *n*, where *n* is a non-zero value for DPSEGSZ.
not specified	not specified	not specified	segmented table space with an implicit specification of SEGSIZE 4

Notes

Segmented table spaces:

If neither LOB, NUMPARTS, nor SEGSIZE are specified, the table space that is created is a segmented table space. See Types of DB2 table spaces (Introduction to DB2 for z/OS) for a discussion of types of table spaces.

Universal table spaces:

If NUMPARTS and SEGSIZE are specified, the table space that is created is a range-partitioned universal table space. If MAXPARTITIONS or MAXPARTITIONS and SEGSIZE are specified, the table space that is created is a partition-by-growth universal table space. If a range-partitioned universal table space contains an XML column, the corresponding XML table space will be range-partitioned universal as well. See Universal table spaces (Introduction to DB2 for z/OS) for information about universal table spaces.

MAXROWS value for table spaces that specify LOCKSIZE PAGE:

For a table space that is defined with LOCKSIZE PAGE that has long columns or specifies MAXROWS 8 or less, there will be no wait for read access. For tables with more rows per page, or if all rows are being accessed or updated, there might be a wait for read access.

Table spaces in a work file database:

The following restrictions apply to table spaces created in a work file database:

- They can be created for another member only if both the executing DB2 subsystem and the other member can access the work file data sets. That is required whether the data sets are user-managed or in a DB2 storage group.
- They cannot use 8 KB or 16 KB page sizes. (The buffer pool in which you define the table space determines the page size. For example, a table space that is defined in a 4 KB buffer pool has 4 KB page sizes.)
- When you create a table space in a work file database, the following clauses are not allowed:

CCSID	LARGE	MEMBER CLUSTER
COMPRESS	LOB	NOT LOGGED
DEFINE NO	LOCKPART	NUMPARTS[1]
DSSIZE[1]	LOCKSIZE	PCTFREE
FREEPAGE	LOGGED	TRACKMOD
GBPCACHE	MAXROWS	

1: DSSIZE and NUMPARTS must not be specified for a table space in a work file database unless MAXPARTITIONS is also specified.

Table spaces for declared temporary tables:

Declared temporary tables and sensitive static scrollable cursors must reside in segmented table spaces in the work file database. At least one table space with a 32KB page size must exist in the work file database before a declared temporary table can be defined and used or before sensitive static scrollable cursors are opened.

Table spaces in the work file database are shared by work files, created and declared global temporary tables and sensitive static scrollable cursor result tables. You cannot specify which table space is to be used for any specific object.

When you create table spaces in the work file database, it is recommended that you give each table space the same segment size, with the same minimum primary and secondary space allocation values for the data sets, to maximize the use of all the table spaces for all objects in all application processes.

Creating LOB table spaces:

When you create a LOB table space, the following clauses are not allowed:

CCSID	LOCKSIZE PAGE	PCTFREE
COMPRESS	LOCKSIZE ROW	SEGSIZE
FREEPAGE	MAXPARTITIONS	TRACKMOD
LOCKSIZE TABLE	NUMPARTS	

Recommended GBPCACHE setting for LOB table spaces:
For LOB table spaces, use the GBPCACHE CHANGED option instead of the GBPCACHE SYSTEM option. Due to the usage patterns of LOBs, the use of GBPCACHE CHANGED can help avoid excessive and synchronous writes to disk and the group buffer pool.

Altering the logging attribute of a table space:
See "Notes" on page 1090 of "ALTER TABLESPACE" on page 1074 for information about altering the logging attributes of a table space.

Table space row formats:
Depending on the value of the RRF subsystem parameter, newly created table spaces will be in either re-ordered row format or basic row format. When the value of the RRF parameter is ENABLE, table spaces will be created in re-ordered row format. When the value of the RRF parameter is DISABLE, newly created table spaces will be created in basic row format. This includes universal table spaces, except for XML table spaces, which are always created in re-ordered row format, regardless of the value of the RRF parameter.

Making a partitioned table space larger:
Depending on the needs of your application, you might need to increase the size of a partitioned table space to hold more data by either adding more partitions or by increasing the size of the existing partitions:

- To add more partitions, use the ALTER TABLE statement with the ADD PARTITION clause.
- To increase the size of the partitions, use the following steps:
  - If the table space is a range-partitioned universal table space, specify a larger DSSIZE using the ALTER TABLESPACE statement if the DSSIZE of the table space is not already at the maximum.
  - If the table space is not a range-partitioned universal table space:
    1. Convert the table space to a range-partitioned universal table space by specifying a SEGSIZE value and a NUMPARTS value using the ALTER TABLESPACE statement
    2. Run the REORG utility with the SHRLEVEL CHANGE or SHRLEVEL REFERENCE option on the table space to apply the new SEGSIZE value
    3. Specify a larger DSSIZE using the ALTER TABLESPACE statement

Related links:

"ALTER TABLESPACE" on page 1074

REORG TABLESPACE (DB2 Utilities)

Redistributing data between existing partitions:
If you need to redistribute the data between the existing partitions to make better use of the space within the existing table, you can use either of these two methods:

- Use the ALTER TABLE statement with the ALTER PARTITION clause. You can alter the partitions to specify new partition boundaries to explicitly specify how to redistribute the data. Any affected partitions are set to REORG-pending status.
- Use the REORG utility with the REBALANCE keyword. REBALANCE specifies that the data is evenly redistributed across the partitions that are reorganized. SeeREORG TABLESPACE (DB2 Utilities) for information about using the REORG utility.

Rules for primary and secondary space allocation:

You can specify the primary and secondary space allocation for table spaces and indexes or allow DB2 to choose them. Having DB2 choose the values, especially the secondary space quantity, increases the possibility of reaching the maximum data set size before running out of extents.

In the following rules that describe how allocation works, these terms are used:

PRIQTY, SECQTY
The keywords for CREATE TABLESPACE, ALTER TABLESPACE, CREATE INDEX, and ALTER INDEX.

specified-priqty
The user-specified value for PRIQTY.

specified-secqty
The user-specified value for SECQTY.

actual-priqty
The actual primary space allocation, in kilobytes.

actual-priqty-cylinders
The actual primary space allocation, in cylinders.

actual-secqty
The actual secondary space allocation, in kilobytes.

actual-secqty-cylinders
The actual secondary space allocation, in cylinders.

calculated-extent-cylinders
A value that is calculated by DB2 using a *sliding scale*. A sliding scale means that the first secondary extent allocations are smaller than later secondary allocations. For example, Figure 19 on page 1477 shows the sliding scale of secondary extent allocations that DB2 uses for a 64-GB data set. The size of each secondary extent is larger for each secondary extent that is allocated up to the 127th extent. For the 127th secondary extent and any subsequent extents, the secondary size allocation is 559 cylinders.

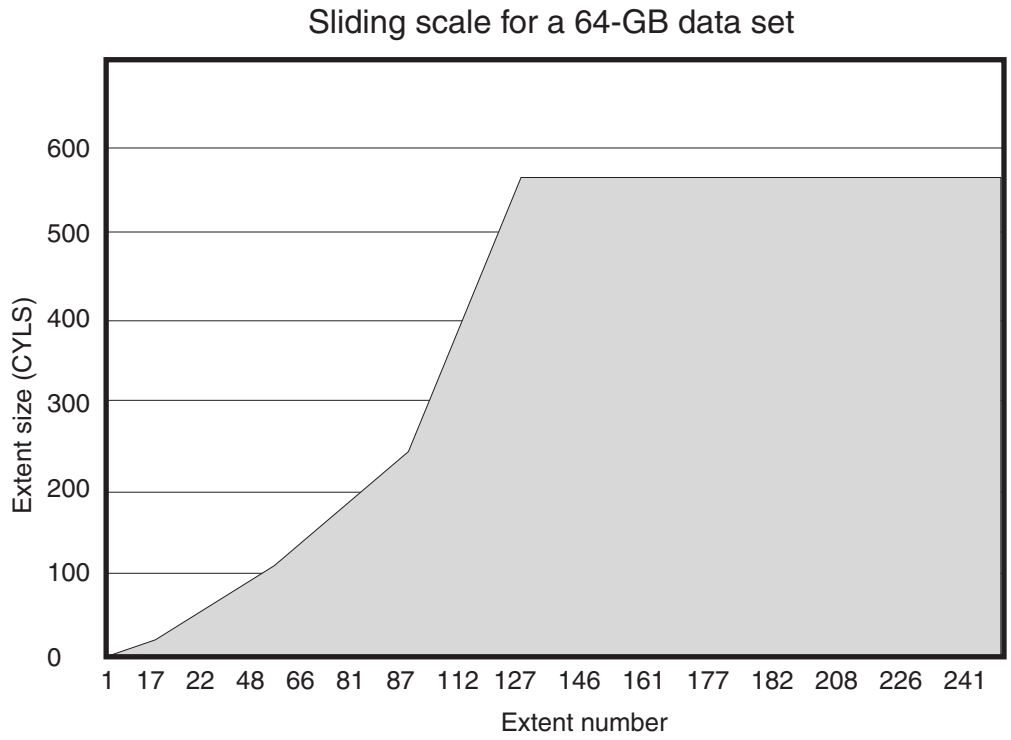


Figure 19. Sliding scale allocation of secondary extents for a 64 GB data set

The rules are:

- **Rule 1 (for primary space allocation)**

  If PRIQTY is specified and *specified-priqty* is not equal to -1, *actual-priqty* is at least *specified-priqty* KB.

  If PRIQTY is not specified or *specified-priqty* is equal to -1, *actual-priqty* is determined as follows:

  – For a table space, if the TSQTY subsystem parameter value is specified and is greater than 0, *actual-priqty* is at least the value of TSQTY.

    If the TSQTY subsystem parameter is not specified or is 0, *actual-priqty* is one cylinder for a non-LOB table space. *actual-priqty* is 10 cylinders for a LOB table space.

  – For an index, if the IXQTY subsystem parameter value is specified and is greater than 0, *actual-priqty* is at least the value of IXQTY.

    If the IXQTY subsystem parameter is not specified or is 0, *actual-priqty* is one cylinder.

- **Rule 2 (for secondary space allocation)**

  If SECQTY is not specified, the following formulas determine *actual-secqty*:

  – If the maximum size of a data set in the table space or index is less than 32 GB, the formula is:

    ```
    actual-secqty-cylinders=
    MAX(0.1*actual-priqty-cylinders, MIN(calculated-extent-cylinders, 127))
    ```

  – If the maximum size of a data set in the table space or index is 32 GB or greater, the formula is:

    ```
    actual-secqty-cylinders=
    MAX(0.1*actual-priqty-cylinders, MIN(calculated-extent-cylinders, 559))
    ```

- **Rule 3 (for secondary space allocation)**

  If SECQTY is 0, *actual-secqty* is 0.
- **Rule 4 (for secondary space allocation)**

  This is the only rule that depends on the value of subsystem parameter MGEXTSZ (field OPTIMIZE EXTENT on installation panel DSNTIP7).

  If MGEXTSZ is YES:

  – If SECQTY is specified and *specified-secqty* is not equal to -1 or 0, the following formulas determine *actual-secqty*:

    - If the maximum size of a data set in the table space or index is less 32 GB, the formula is:

      ```
      actual-secqty-cylinders=
      MAX(MIN(calculated-extent-cylinders, 127),specified-secqty-cylinders)
      ```

    - If the maximum size of a data set in the table space or index is 32 GB or greater, the formula is:

      ```
      actual-secqty-cylinders=
      MAX(MIN(calculated-extent-cylinders, 559),specified-secqty-cylinders)
      ```

  If MGEXTSZ is NO:

  – For a table space, if SECQTY is *n*, the secondary space allocation is at least *n* kilobytes, with the following exceptions:

    - If SECQTY is greater than 4194304, *n* is 4194304 kilobytes.
    - For LOB table spaces:
      - For 4KB page sizes, if *integer* is greater than 0 and less than 200, *n* is 200.
      - For 8KB page sizes, if *integer* is greater than 0 and less than 400, *n* is 400.
      - For 16KB page sizes, if *integer* is greater than 0 and less than 800, *n* is 800.
      - For 32KB page sizes, if *integer* is greater than 0 and less than 1600, *n* is 1600.
      - For any page size, if *integer* is greater than 4194304, *n* is 4194304.

  – For an index, if SECQTY is *integer*, the secondary space allocation is at least *n* kilobytes, where *n* is:

    12 If SECQTY and PRIQTY are omitted

    4194304
    If *integer* is greater than 4194304

    integer If *integer* is not greater than 4194304
- **Rule 5 (for secondary space allocation):** When a table space requires a new piece, the primary allocation quantity of the new piece is determined as follows:

  – If the value of subsystem parameter MGEXTSZ is NO, the primary quantity from rule 1 is used.

  – Otherwise, the maximum of the following values is used:

    - The quantity that is calculated through sliding scale methodology
    - The primary quantity from rule 1
    - The specified SECQTY value

Alternative syntax and synonyms:

For compatibility with previous releases of DB2, the following keywords are supported:

- You can specify the LOCKPART clause, but it has no effect. DB2 treats all table spaces as if they were defined as LOCKPART YES. LOCKPART YES specifies the use of selective partition locking. When all the conditions for selective partition locking are met, DB2 locks only the partitions that are accessed. When the conditions for selective partition locking are not met, DB2 locks every partition of the table space.

  LOCKSIZE TABLESPACE and LOCKPART YES are mutually exclusive.
- When creating a partitioned table space, you can specify PART as a synonym for PARTITION.
- When specifying the logging attributes for a table space, you can specify LOG YES as a synonym for LOGGED, and you can specify LOG NO as a synonym for NOT LOGGED.
- You can specify the LARGE clause when creating partitioned table spaces, but DSSIZE is the preferred clause to use when specifying the partition size.

Although these keywords are supported as alternatives, they are not the preferred syntax.

Examples

Example 1: Create table space DSN8S11D in database DSN8D11A. Let DB2 define the data sets, using storage group DSN8G110. The primary space allocation is 52 kilobytes; the secondary, 20 kilobytes. The data sets need not be erased before they are deleted.

Locking on tables in the space is to take place at the page level. Associate the table space with buffer pool BP1. The data sets can be closed when no one is using the table space.

```
CREATE TABLESPACE DSN8S11D
  IN DSN8D11A
  USING STOGROUP DSN8G110
    PRIQTY 52
    SECQTY 20
    ERASE NO
  LOCKSIZE PAGE
  BUFFERPOOL BP1
  CLOSE YES;
```

For the above example, the underlying data sets for the table space will be created immediately, which is the default (DEFINE YES). If you want to defer the creation of the data sets until data is first inserted into the table space, you would specify DEFINE NO instead of accepting the default behavior.

Example 2: Assume that a large query database application uses a table space to record historical sales data for marketing statistics. Create large table space SALESHX in database DSN8D11A for the application. Create it with 82 partitions, specifying that the data in partitions 80 through 82 is to be compressed.

Let DB2 define the data sets for all the partitions in the table space, using storage group DSN8G110. For each data set, the primary space allocation is 4000 kilobytes, and the secondary space allocation is 130 kilobytes. Except for the data set for partition 82, the data sets do not need to be erased before they are deleted.

Locking on the table is to take place at the page level. There can only be one table in a partitioned table space. Associate the table space with buffer pool BP1. The

data sets cannot be closed when no one is using the table space. If there are no CLOSE YES data sets to close, DB2 might close the CLOSE NO data sets when the DSMAX is reached.

```
CREATE TABLESPACE SALESHX
  IN DSN8D11A
  USING STOGROUP DSN8G110
    PRIQTY 4000
    SECQTY 130
    ERASE NO
  NUMPARTS 82
  (PARTITION 80
    COMPRESS YES,
   PARTITION 81
    COMPRESS YES,
   PARTITION 82
    COMPRESS YES
    ERASE YES)
  LOCKSIZE PAGE
  BUFFERPOOL BP1
  CLOSE NO;
```

Example 3: Assume that a column named EMP_PHOTO with a data type of BLOB(110K) has been added to the sample employee table for each employee's photo. Create LOB table space PHOTOLTS in database DSN8D11A for the auxiliary table that will hold the BLOB data.

Let DB2 define the data sets for the table space, using storage group DSN8G110. For each data set, the primary space allocation is 3200 kilobytes, and the secondary space allocation is 1600 kilobytes. The data sets do not need to be erased before they are deleted. (Because ERASE NO is the default, the clause does not have to be explicitly specified to get that behavior.)

```
CREATE LOB TABLESPACE PHOTOLTS
  IN DSN8D11A
  USING STOGROUP DSN8G110
    PRIQTY 3200
    SECQTY 1600
  LOCKSIZE LOB
  BUFFERPOOL BP16K0
  GBPCACHE SYSTEM
  NOT LOGGED
  CLOSE NO;
```

Example 4: The following example creates a range-partitioned universal table space, TS1, in database DSN8D11A using storage group DSN8G110. The table space has 16 pages per segment and has 55 partitions. It specifies LOCKSIZE ANY.

```
CREATE TABLESPACE TS1
  IN DSN8D11A
  USING STOGROUP DSN8G110
  NUMPARTS 55
  SEGSIZE 16
  LOCKSIZE ANY;
```

Example 5: The following example creates a range-partitioned universal table space, TS2, in database DSN8D11A using storage group DSN8G110. The table space has 64 pages per segment and has seven defer-defined partitions, where every other partition is compressed.

```
CREATE TABLESPACE TS2
  IN DSN8D11A
  USING STOGROUP DSN8G110
  NUMPARTS 7
  (
```

```
 PARTITION 1 COMPRESS YES,
 PARTITION 3 COMPRESS YES,
 PARTITION 5 COMPRESS YES,
 PARTITION 7 COMPRESS YES
)
SEGSIZE 64
DEFINE NO;
```

Example 6: The following example creates a partition-by-growth table space that has a maximum size of 2 GB for each partition, four pages per segment with a maximum of 24 partitions for the table space.

```
CREATE TABLESPACE TS01TS IN TS01DB USING STOGROUP SG1
  DSSIZE 2G
  MAXPARTITIONS 24
  LOCKSIZE ANY
  SEGSIZE 4;
```

CREATE TRIGGER

The CREATE TRIGGER statement defines a trigger in a schema and builds a trigger package at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set that is defined below must include at least one of the following:

- The CREATEIN privilege on the schema
- SYSADM or SYSCTRL authority
- System DBADM

In defining a trigger on a table, the privilege set that is defined below must include SYSADM authority or each of the following:

- The SELECT privilege on the table on which the trigger is defined if any transition variables or transition tables are specified
- The SELECT privilege on any table or view to which the search condition of triggered action refers
- The necessary privileges to invoke the triggered SQL statements in the triggered action
- The authorization to define a trigger on the table, which must include at least one of the following:
  - The TRIGGER privilege on the table on which the trigger is defined
  - The ALTER privilege on the table on which the trigger is defined
  - DBADM authority on the database that contains the table
  - SYSCTRL authority

  If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

In defining a trigger on a view, the privilege set that is defined below must include SYSADM authority or each of the following:

- The SELECT privilege on any table or view to which the search condition of triggered action refers
- The necessary privileges to invoke the triggered SQL statements in the triggered action
- The authorization to define a trigger on the view, which must include at least one of the following:
  - Ownership of the view on which the trigger is defined
  - SYSCTRL authority

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the owner is a role, the implicit schema match does not apply and this role needs to include one of the previously listed conditions.

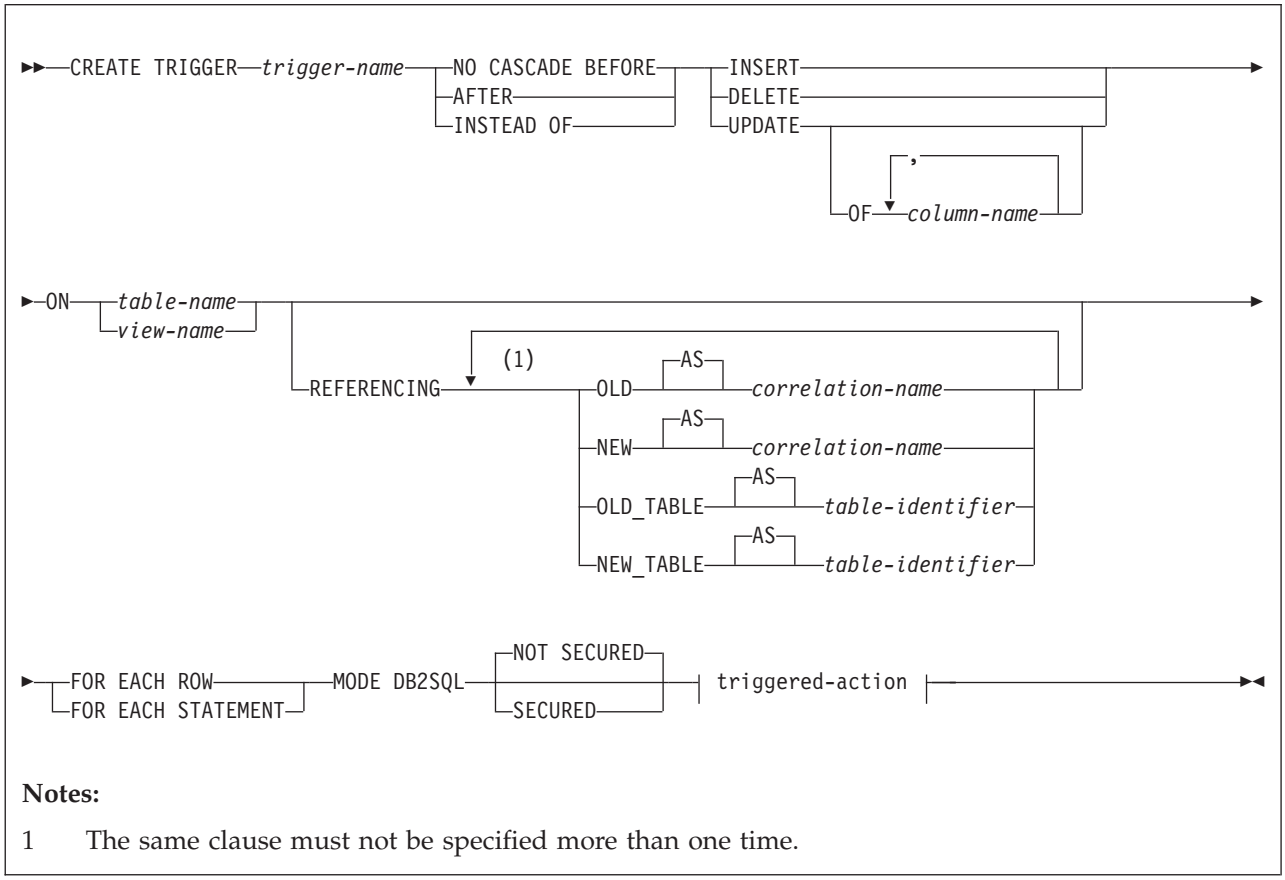
If the statement is dynamically prepared and is not running in a trusted context for which the ROLE AS OBJECT OWNER clause is specified, the privilege set is the set of privileges that are held by the SQL authorization ID of the process. The specified trigger name can include a schema name (a qualifier). If the schema name is not the same as the SQL authorization ID of the process, one of the following conditions must be met:

- The privilege set includes SYSADM or SYSCTRL authority.
- The SQL authorization ID of the process has the CREATEIN privilege on the schema.

At least one of the following privileges is required if the SECURED option is specified:

- SECADM authority
- CREATE_SECURE_OBJECT privilege

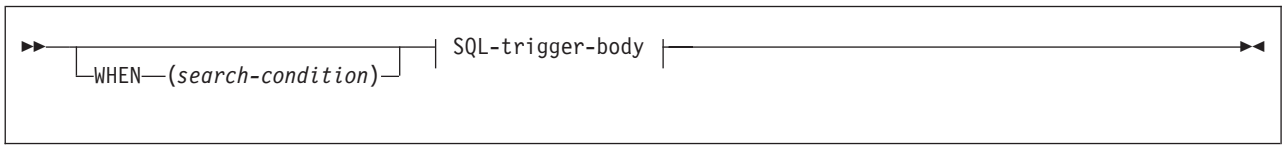
Syntax

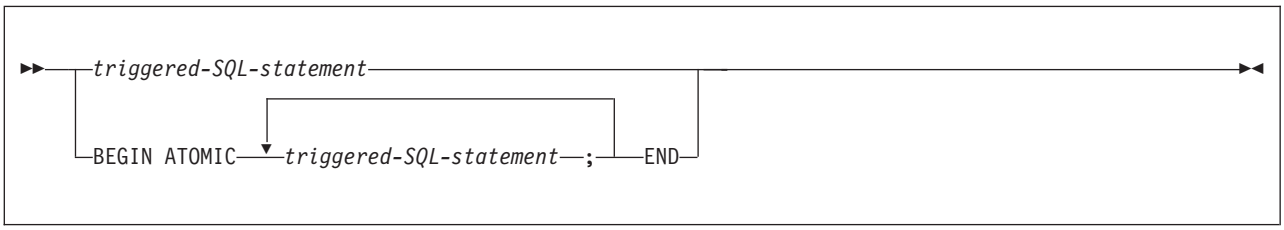


Notes:

1 The same clause must not be specified more than one time.

triggered-action





Description

trigger-name
Names the trigger. The name, including the implicit or explicit schema name, must not identify a trigger that exists at the current server.

The name is also used to create the trigger package; therefore, the name must also not identify a package that is already described in the catalog. The schema name becomes the collection-id of the trigger package. Although *trigger-name* can be specified as an ordinary or delimited identifier, the name should conform to the rules for an ordinary identifier. Refer to The implicitly created trigger package for additional information.

The schema name must not begin with 'SYS' unless the name is 'SYSADM', or the schema name is 'SYSTOOLS' and the user who executes the CREATE statement has SYSADM or SYSCTRL privilege.

NO CASCADE BEFORE
Specifies that the trigger is a before trigger. DB2 executes the triggered action before it applies any changes caused by an insert, delete, or update operation on the subject table. It also specifies that the triggered action does not activate other triggers because the triggered action of a before trigger cannot contain any updates.

NO CASCADE BEFORE must not be specified when *view-name* is also specified. FOR EACH ROW must be specified for a BEFORE trigger.

AFTER
Specifies that the trigger is an after trigger. DB2 executes the triggered action after it applies any changes caused by an insert, delete, or update operation on the subject table. AFTER must not be specified if *view-name* is also specified.

INSTEAD OF
Specifies that the trigger is an instead of trigger. The associated triggered action replaces the action against the subject view. Only one INSTEAD OF trigger is allowed for each type of operation on a given subject view. DB2 executes the triggered-action instead of the insert, update, or delete operation on the subject view.

INSTEAD OF must not be specified when *table-name* is also specified. The WHEN clause can not be specified for an INSTEAD OF trigger. FOR EACH STATEMENT must not be specified for an INSTEAD OF trigger.

INSERT
Specifies that the trigger is an insert trigger. DB2 executes the triggered action whenever there is an insert operation on the subject table. However, if the insert trigger is defined on any explain table, and the insert operation was caused by DB2 adding a row to the table, the triggered action is not to be executed.

DELETE
Specifies that the trigger is a delete trigger. DB2 executes the triggered action whenever there is a delete operation on the subject table.

UPDATE
Specifies that the trigger is an update trigger. DB2 executes the triggered action whenever there is an update operation on the subject table.

If you do not specify a list of column names, an update operation on any column of the subject table, including columns that are subsequently added with the ALTER TABLE statement, activates the triggered action.

OF *column-name,...*
Each *column-name* that you specify must be a column of the subject table and must appear in the list only once. An update operation on any of the listed columns activates the triggered action.

UPDATE OF *column-name* cannot be specified for an INSTEAD OF trigger.

ON *table-name*
Identifies the subject table of the BEFORE or AFTER trigger definition. The name must identify a base table that exists at the current server. It must not identify a materialized query table, a clone table, a temporary table, an auxiliary table, an alias, a synonym, a real-time statistics table, or a catalog table.

ON *view-name*
Identifies the subject view of the INSTEAD OF trigger definition. The name must identify a view that exists at the current server.

view-name must not specify a view where any of the following conditions are true:

- The view is defined with the WITH CASCADED CHECK option (a symmetric view)
- The view on which a symmetric view has been defined
- The view references data that is encoded with different encoding schemes or CCSID values
- The view has a column that is a ROWID column
- The view has a column that is based on an underlying column of any of the following types:
  - A security label column
  - A row change timestamp column
  - A ROW BEGIN column
  - A ROW END column
  - A transaction start ID column
- The view has a column that is defined (directly or indirectly) as an expression
- The view has a column that is based on a column of a result table that involves a set operator
- The view has columns that have field procedures
- All of the underlying tables of the view are catalog tables
- All of the underlying tables of the view are created global temporary tables
- All of the underlying tables of the view are clone tables
- The view has other views that are dependent on it

REFERENCING
Specifies the correlation names for the transition variables and the table names

for the transition tables. For the rows in the subject table that are modified by the triggering SQL operation (insert, delete, or update), a correlation name identifies the columns of a specific row. *table-identifiers* identify the complete set of affected rows. Transition variables with XML types cannot be referenced inside of a trigger. If the column of a transition table is referenced, the data type of the column cannot be XML.

Each row that is affected by the triggering SQL operation is available to the triggered action by qualifying column names with *correlation-names* that are specified as follows:

OLD AS *correlation-name*
Specifies the correlation name that identifies the values in the row prior to the triggering SQL operation.

NEW AS *correlation-name*
Specifies the correlation name that identifies the values in the row as modified by the triggering SQL operation and by any SET statement in a before trigger that has already been executed.

The complete set of rows that are affected by the triggering operation is available to the triggered action by using *table-identifiers* that are specified as follows:

OLD_TABLE AS *table-identifier*
Specifies the name of a temporary table that identifies the values in the complete set of rows that are modified rows by the triggering SQL operation prior to any actual changes.

NEW_TABLE AS *table-identifier*
Specifies the name of a temporary table that identifies the values in the complete set of rows as modified by the triggering SQL operation and by any SET statement in a before trigger that has already been executed.

Only one OLD and one NEW *correlation-name* can be specified for a trigger. Only one OLD_TABLE and one NEW_TABLE *table-identifier* can be specified for a trigger. All of the *correlation-names* and *table-identifiers* must be unique from one another.

Table 127 on page 1487 summarizes the allowable combinations of transition variables and transition tables that you can specify for the various trigger types. The OLD *correlation-name* and the OLD_TABLE *table-identifier* are valid only if the triggering event is either a delete operation or an update operation. For a delete operation, the OLD *correlation-name* captures the values of the columns in the deleted row, and the OLD_TABLE *table-identifier* captures the values in the set of deleted rows. For an update operation, the OLD *correlation-name* captures the values of the columns of a row before the update operation, and the OLD_TABLE *table-identifier* captures the values in the set of updated rows.

The NEW *correlation-name* and the NEW_TABLE *table-identifier* are valid only if the triggering event is either an insert operation or an update operation. For both operations, the NEW *correlation-name* captures the values of the columns in the inserted or updated row and the NEW_TABLE *table-identifier* captures the values in the set of inserted or updated rows. For BEFORE triggers, the values of the updated rows include the changes from any SET statements in the triggered action of BEFORE triggers.

The OLD and NEW *correlation-name* variables cannot be modified in an AFTER or INSTEAD OF trigger.

Table 127. Allowable combinations of attributes in a trigger definition

Granularity	Activation time	Triggering SQL operation	Transition variables allowed1	Transition tables allowed1
FOR EACH ROW	BEFORE	DELETE	OLD	None
		INSERT	NEW	None
		UPDATE	OLD, NEW	None
	AFTER	DELETE	OLD	OLD_TABLE
		INSERT	NEW	NEW_TABLE
		UPDATE	OLD, NEW	OLD_TABLE, NEW_TABLE
	INSTEAD OF	DELETE	OLD	OLD_TABLE
		INSERT	NEW	NEW_TABLE
		UPDATE	OLD, NEW	OLD_TABLE, NEW_TABLE
FOR EACH STATEMENT	AFTER	DELETE	None	OLD_TABLE
		INSERT	None	NEW_TABLE
		UPDATE	None	OLD_TABLE, NEW_TABLE

Note:

1. If a transition table or variable is referenced where it is not allowed, an error is returned.

A transition variable that has a character data type inherits the subtype and CCSID of the column of the subject table. During the execution of the triggered action, the transition variables are treated like host variables. Therefore, character conversion might occur. However, unlike a host variable, a transition variable can have the bit data attribute, and character conversion never occurs for bit data. A transition variable is considered to be bit data if the column of the table to which it corresponds is bit data.

You cannot modify a transition table; transition tables are read-only. Although a transition table does not inherit any edit or validation procedures from the subject table, it does inherit the encoding scheme and field procedures of the subject table.

The scope of each *correlation-name* and each *table-identifier* is the entire trigger definition.

FOR EACH ROW or FOR EACH STATEMENT

Specifies the conditions for which DB2 executes the triggered action.

FOR EACH ROW

Specifies that DB2 executes the triggered action for each row of the subject table that the triggering SQL operation modifies. If the triggering SQL operation does not modify any rows, the triggered action is not executed.

FOR EACH STATEMENT

Specifies that DB2 executes the triggered action only one time for the triggering operation. Even if the triggering operation does not modify or delete any rows, the triggered action is executed one time.

FOR EACH STATEMENT must not be specified for a BEFORE or INSTEAD OF trigger.

MODE DB2SQL
Specifies the mode of the trigger. MODE DB2SQL triggers are activated after all of the row operations have occurred.

NOT SECURED or SECURED
Specifies whether the trigger is considered secure. **NOT SECURED** is the default.

SECURED
Specifies the trigger is considered secure.

SECURED must be specified for a trigger if its subject table is using row access control or column access control. SECURED must also be specified for a trigger that is created for a view and one or more of the underlying tables in the view definition is using row access control or column access control.

NOT SECURED
Specifies the trigger is considered not secure.

NOT SECURED must not be specified for a trigger whose subject table is using row access control or column access control. NOT SECURED must also not be specified for a trigger that is created for a view and one or more of the underlying tables in the view definition is using row access control or column access control.

triggered-action
Specifies the action to be performed when the trigger is activated. The *triggered-action* is composed of one or more SQL statements and an optional condition that controls whether the statements are executed.

WHEN (*search-condition***)**
Specifies a condition that evaluates to true, false, or unknown. The triggered SQL statements are executed only if the *search-condition* evaluates to true. If the WHEN clause is omitted, the associated SQL statements are always executed. The condition for a before trigger must not include a subselect that references the subject table.

The WHEN clause must not be specified for an INSTEAD OF trigger.

search-condition must not reference the following items:

- A system-period temporal table if the trigger package is generated with the SYSTIMESENSITIVE(YES) bind option
- An archive-enabled table if the trigger package is generated with the ARCHIVESENSITIVE(YES) bind option

SQL-trigger-body
Specifies the SQL statements that are to be executed for the triggered action.

triggered-SQL-statement
Specifies a single SQL statement that is to be executed for the triggered action.

BEGIN ATOMIC *triggered-SQL-statement*,... **END**
Specifies a list of SQL statements that are to be executed for the triggered action. The statements are executed in the order in which they are specified.

SQL processor programs, such as SPUFI, the command line processor, and DSNTEP2, might not correctly parse SQL statements in the triggered action that are ended with semicolons. These processor

programs accept multiple SQL statements, each separated with a terminator character, as input. Processor programs that use a semicolon as the SQL statement terminator can truncate a CREATE TRIGGER statement with embedded semicolons and pass only a portion of it to DB2. Therefore, you might need to change the SQL terminator character for these processor programs. For information on changing the terminator character for SPUFI and DSNTEP2, see *DB2 Application Programming and SQL Guide*.

Only certain SQL statements can be specified in the *SQL-trigger-body*. Table 128 shows the list of allowable SQL statements, which differs depending on whether the trigger is being defined as BEFORE, AFTER, or INSTEAD OF. An 'X' in the table indicates that the statement is valid.

Table 128. Allowable SQL statements

	Trigger activation time		
SQL statement	**BEFORE**	**AFTER**	**INSTEAD OF**
CALL	X	X	X
DELETE (searched)		X	X
fullselect	X	X	X
INSERT		X	X
MERGE		X	X
REFRESH TABLE		X	X
SET transition variable	X		
SIGNAL	X	X	X
UPDATE (searched)		X	X
VALUES	X	X	X

The statements in the triggered action have these restrictions:

- They must not refer to host variables, parameter markers, undefined transition variables, or declared temporary tables.
- They must only refer to a table or view that is at the current server.
- They must only invoke a stored procedure or user-defined function that is at the current server. An invoked routine can, however, access a server other than the current server.
- They must not contain a fullselect that refers to the subject table if the trigger is defined as BEFORE.
- They must not modify a column that is part of a BUSINESS_TIME period.

The triggered action can refer to the values in the set of affected rows. This action is supported through the use of transition variables and transition tables.

Transition variables use the names of the columns in the subject table qualified by a specified name that identifies whether the reference is to the old value (before the update) or the new value (after the update). A transition variable can be referenced in *search-condition* or *triggered-SQL-statement* of the triggered action wherever a host variable is allowed in the statement if it were issued outside the body of a trigger.

Transition tables can be referenced in the triggered action of an after trigger. Transition tables are read-only. Transition tables also use the name of the columns of the subject table but have a name specified that allows the complete set of affected rows to be treated as a table. The name of the transition table can be referenced in *triggered-SQL-statement* of the triggered action whenever a table name is allowed in the statement if it were issued outside the body of a trigger. The name of the transition table can be specified in *search-condition* or *triggered-SQL-statement* of the triggered action whenever a column name is allowed in the statement if it were issued outside the body of a trigger.

In addition, a transition table can be passed as a parameter to a user-defined function or procedure specifying the TABLE keyword before the name of the transition table. When the function or procedure is invoked, a table locator is passed for the transition table.

A transition variable or transition table is not affected after being returned from a procedure invoked from within a triggered action regardless of whether the corresponding parameter was defined in the CREATE PROCEDURE statement as IN, INOUT, or OUT.

Notes

Owner privileges:

When an INSTEAD OF trigger is defined, the associated privilege (INSERT, UPDATE, or DELETE on the view) is given to the owner of the view. The owner is granted the privilege with the ability to grant that privilege to others. For more information about ownership of an object, see "Authorization, privileges, permissions, masks, and object ownership" on page 70.

Execution authorization:

The user executing the triggering SQL operation does not need authority to execute a triggered-SQL-statement. A triggered-SQL-statement will execute using the authority of the owner of the trigger.

Activating a trigger:

Only insert, delete, or update operations can activate a trigger. The activation of a trigger might cause trigger cascading. *Trigger cascading* is the result of the activation of one trigger that executes SQL statements that cause the activation of other triggers or even the same trigger again. The triggered actions might also cause updates as a result of the original modification, which can result in the activation of additional triggers. With trigger cascading, a significant chain of triggers might be activated, causing a significant change to the database as a result of a single insert, delete, or update operation.

Loading a table with the LOAD utility does not activate any triggers that are defined for the table if SHRLEVEL NONE is specified as part of the LOAD utility or if the default SHRLEVEL option for LOAD is taken. If SHRLEVEL CHANGE is specified as part of the LOAD utility, triggers are activated when loading a table with the LOAD utility.

Adding triggers to enforce constraints:

Adding a trigger on a table that already has rows in it will not cause the triggered action to be executed. Thus, if the trigger is designed to enforce constraints on the data in the table, the data in the existing rows might not satisfy those constraints.

Multiple triggers:

Multiple triggers that have the same triggering SQL operation and activation time can be defined on a table. The triggers are activated in the order in which they were created. For example, the trigger that was created first is executed first; the trigger that was created second is executed second.

Read-only views:

The addition of an INSTEAD OF trigger for a view affects the read only characteristic of the view. If a read-only view has a dependency relationship with an INSTEAD OF trigger, the type of operation that is defined for the INSTEAD OF trigger defines whether the view is deletable, insertable, or updatable.

The creation of an INSTEAD OF trigger causes dependent packages, plans, and statements in the dynamic statement cache to be marked invalid if the view definition is not read-only.

Invalidation of plans and packages:

A trigger package becomes invalid if an object or privilege on which it depends is dropped or revoked. The next time the trigger is activated, DB2 attempts to rebind the invalid trigger package. If the automatic rebind is unsuccessful, the trigger package remains invalid.

You cannot create another package from the trigger package, such as with the BIND COPY command. The only way to drop a trigger package is to drop the trigger or the subject table. Dropping the trigger drops the trigger package; dropping the subject table drops the trigger and the trigger package.

DB2 creates the trigger package with the following initial attributes (some of these attributes can be modified using the REBIND TRIGGER PACKAGE command):

- ACTION(ADD)
- ARCHIVESENSITIVE(YES)
- BUSTIMESENSITIVE(YES)
- CURRENTDATA(NO)
- DBPROTOCOL(DRDA)
- DEGREE(1)
- DESCSTAT(value from the DESCSTAT subsystem parameter)
- DYNAMICRULES(BIND)
- ENABLE(*)
- ENCODING(0)
- EXPLAIN(NO)
- FLAG(I)
- ISOLATION(CS)
- REOPT(NONE) and NODEFER(PREPARE)
- OPTHINT
- OWNER(authorization ID) or ROLE of appropriate
- PATH(path)
- RELEASE(COMMIT)
- ROUNDING(value from the CURRENT DECFLOAT ROUNDING MODE special register)

- SQLERROR(NOPACKAGE)
- SYSTIMESENSITIVE(YES)
- QUALIFIER(authorization ID)
- VALIDATE(BIND)

The values of OWNER, QUALIFIER, and PATH are set depending on whether the CREATE TRIGGER statement is embedded in a program or issued interactively. If the statement is embedded in a program, OWNER and QUALIFIER are the owner and qualifier of the package or plan. PATH is the value from the PATH bind option. If the statement is issued interactively, both OWNER and QUALIFIER are the SQL authorization ID. PATH is the value in the CURRENT PATH special register.

Transition variable values and INSTEAD OF triggers:

The initial values for new transition variables or new transition table columns that are visible in an INSTEAD OF INSERT trigger are set as follows:

- If a value is explicitly specified for a column in the insert operation, the corresponding new transition variable is that explicitly specified value.
- If a value is not explicitly specified for a column in the insert operation or the DEFAULT clause is specified, the corresponding new transition variable is:
  - the default value of the underlying table column if the view column is updatable (without the INSTEAD OF trigger)
  - otherwise, the null value

  If a view column is not nullable and does not have a default, the value must be explicitly specified in the insert operation.

The initial values for new transition variables that are visible in an INSTEAD OF UPDATE trigger are set as follows:

- If a value is explicitly specified for a column in the update operation, the corresponding new transition variable is that explicitly specified value
- If the DEFAULT clause is explicitly specified for a column in the update operation, the corresponding new transition variable is:
  - the default value of the underlying table column if the view column is updatable (without the INSTEAD OF trigger)
  - otherwise, the null value

  If a view column is not nullable and does not have a default, the value must be explicitly specified in the update operation.
- Otherwise, the corresponding new transition variable is the existing value of the column in the row.

Considerations for a MERGE statement:

The MERGE statement can execute insert and update operations. The applicable INSERT or UPDATE triggers are activated for the MERGE statement when an insert or update operation is executed.

Considerations for implicitly hidden columns:

In the body of a trigger, a trigger transition variable that corresponds to an implicitly hidden column can be referenced. A trigger transition table, that corresponds to a table with an implicitly hidden column, includes that column as part of the transition table. Likewise, a trigger transition variable will exist for the column that is defined as implicitly hidden. A trigger transition variable that corresponds to an implicitly hidden column can be referenced in the body of a trigger.

Considerations for the special plan, statement, and function tables for EXPLAIN:

You can create a trigger on PLAN_TABLE, DSN_STATEMNT_TABLE, or DSN_FUNCTION_TABLE. However, insert triggers that are defined on these tables are not activated when DB2 adds rows to the tables.

Adding columns to a subject table or a table referenced in the *triggered action*:

If a column is added to the subject table after triggers have been defined, the following rules apply:

- If the trigger is an update trigger that was defined without an explicit list of column names, an update to the new column activates the trigger.
- If the subject table is referenced in the *triggered-action*, the new column is not accessible to the SQL statements until the trigger package is rebound.
- The OLD_TABLE and the NEW_TABLE transition tables contain the new column, but the column cannot be referenced unless the trigger is re-created. If the transition tables are passed to a user-defined function or a stored procedure, the user-defined function or stored procedure must be re-created with the new definition of the table (that is, the function or procedure must be dropped and re-created), and the package for the user-defined function or stored procedure must be rebound.

If a column is added to any table that is referenced in the *triggered-action*, the new column is not accessible to the SQL statements until the trigger package is rebound.

Altering the attributes of a column that the *triggered action* references:

If a column is altered in the table on which the trigger is defined (the subject table), the alter is processed, and the dependent trigger packages are invalidated.

Renaming the table for which the trigger is defined, or tables referenced in the triggered action:

You cannot rename a table for which a trigger is defined (the subject table). Except for the subject table, you can rename any table to which the SQL statements in the triggered action refer. After renaming such a table, drop the trigger and then re-create the trigger so that it refers to the renamed table.

Dependencies when dropping objects and revoking privileges:

The following dependencies apply to a trigger:

- Dropping the subject table (the table on which the trigger is defined) causes the trigger and its package to also be dropped.
- Dropping any table, view, alias, or index that is referenced or used within the SQL statements in the triggered action causes the trigger and its package to be invalidated. Dropping a referenced synonym has no effect.
- Dropping a user-defined function that is referenced by the SQL statements in the triggered action is not allowed. An error occurs.
- Dropping a sequence that is referenced by the SQL statements in the triggered action is not allowed. An error occurs.
- Revoking a privilege on which the trigger depends causes the trigger and its package to be invalidated.

Errors executing triggers:

Severe errors that occur during the execution of triggered SQL statements are returned with SQLCODE -901, -906, -911, and -913 and the

corresponding SQLSTATE. Non-severe errors raised by a triggered SQL statement that is a SIGNAL statement or that contains a RAISE_ERROR function are returned with SQLCODE -438 and the SQLSTATE that is specified in the SIGNAL statement or the RAISE_ERROR condition. Other non-severe errors are returned with SQLCODE -723 and SQLSTATE 09000. Warnings are not returned.

Special registers:

The values of the special registers are saved before a trigger is activated and are restored on return from the trigger.

The following table describes how special registers are set on entry to the trigger body. Some of the special registers are applicable only to dynamic SQL. Although dynamic SQL statements are not allowed directly in the triggered SQL statements, they are allowed in a user-defined function or stored procedure that is invoked by the triggered SQL statements.

Table 129. Rules for the values of special registers in triggers

Special register	The value is ....
CURRENT DATE CURRENT TIME CURRENT TIMESTAMP CURRENT TEMPORAL BUSINESS_TIME CURRENT TEMPORAL SYSTEM_TIME	Inherited from the triggering SQL operation (delete, insert, update). All triggered SQL statements, including the SQL statements in a user-defined function or a stored procedure invoked by the trigger, inherit these values.
CURRENT PACKAGESET	Set to the schema name of the trigger
CURRENT TIMEZONE	Set to the TIMEZONE parameter

Table 129. Rules for the values of special registers in triggers (continued)

Special register	The value is ....
• CURRENT CLIENT_ACCTNG • CURRENT CLIENT_APPLNAME • CURRENT CLIENT_USERID • CURRENT CLIENT_WRKSTNNAME • CURRENT APPLICATION ENCODING SCHEME • CURRENT DECFLOAT ROUNDING MODE • CURRENT DEBUG MODE • CURRENT DEGREE • CURRENT GET_ACCEL_ARCHIVE • CURRENT LC_CTYPE • CURRENT MAINTAINED TABLE TYPES • CURRENT MEMBER • CURRENT OPTIMIZATION HINT • CURRENT PATH • CURRENT PACKAGE PATH • CURRENT PRECISION • CURRENT QUERY ACCELERATION • CURRENT REFRESH AGE • CURRENT RULES • CURRENT SERVER • CURRENT SQLID • SESSION_USER	Inherited from the triggering SQL operation (delete, insert, update)

Result sets for stored procedures:

If a trigger invokes a stored procedure that returns result sets, the application that activated the trigger cannot access those result sets.

Transaction isolation:

All of the statements in the *SQL-trigger-body* run under the isolation level in effect for the trigger.

Limiting processor time:

The DB2 resource limit facility allows you to specify the maximum amount of processor time for a dynamic, manipulative SQL statement such as SELECT or SQL data change statements. The execution of a trigger is counted as part of the triggering SQL statement.

Characteristics of the package that is generated for a trigger:

The package that is associated with the trigger is named as follows:

- *location* is set to the value of the CURRENT SERVER special register
- *collection-id* (schema) for the package is the same as the schema qualifier of the trigger
- *package-id* is the same as the name of the trigger
- *version-id* is the same as the empty string

The package is generated with the bind options that correspond to the implicitly or explicitly specified trigger options.

Multiple versions of a trigger package are not allowed. Use the REBIND command to explicitly rebind the trigger package. To specify the name of a trigger package for the bind commands, the trigger name must conform to the rules for an ordinary identifier.

A trigger package becomes invalid if an object or a privilege on which it depends is dropped or revoked. The next time that the trigger is activated, DB2 attempts to rebind the invalid trigger package. If the automatic rebind is unsuccessful, the trigger package remains invalid.

You cannot create another package from the trigger package, such as with the BIND COPY command. The only way to drop a trigger package is to drop the trigger or the subject table. Dropping the trigger drops the trigger package. Dropping the subject table drops the trigger and the trigger package.

Creating a trigger with the SECURED option:

Typically, the security administrator will examine the data that is accessed by a trigger, ensure that it is secure, and grant the CREATE_SECURE_OBJECT privilege to someone who requires the privileges to create a secured trigger. After the trigger is created, the security administrator will revoke the CREATE_SECURE_OBJECT privilege from the owner of the trigger.

The trigger is considered secure after the CREATE TRIGGER statement is executed. DB2 treats the SECURED attribute as an assertion that declares that the user has established an audit procedure for all activities in the trigger body. If a secure trigger references user-defined functions, DB2 assumes those functions are secure without validation. If those functions can access sensitive data, the user with SECADM authority needs to ensure that those functions are allowed to access that data and that an audit procedure is in place for all versions of those functions, and that all subsequent ALTER FUNCTION statements or changes to external packages are being reviewed by this audit process.

A trigger must be secure if its subject table is using row access control or column access control. SECURED must also be specified for a trigger that is created for a view and one or more of the underlying tables in the view definition is using row access control or column access control.

Creating a trigger with the NOT SECURED option:

The CREATE TRIGGER statement returns an error if the subject table of the trigger is using row access control or column access control, or if the trigger is for a view and one or more of the underlying tables in the view definition is using row access control or column access control.

Row and column access control that is not enforced for transition variables and transition tables:

If row access control or column access control is enforced for the subject table of the trigger, row permissions and column masks are not applied to the initial values of transition variables and transition tables. Row access control and column access control is enforced for the triggering table, but is ignored for transition variables and transition tables that are referenced in the body of the trigger body or are passed as arguments to user-defined functions that are invoked in the body of the trigger. To ensure that there are no security concerns for SQL statements accessing sensitive data in transition variables and transition tables in the trigger action, the trigger must be created with the SECURED option. If a trigger is not secure, the CREATE TRIGGER statement returns an error.

The implicitly created trigger package:

When you create a trigger, DB2 automatically creates a trigger package with the same name as the trigger name. The collection name of the trigger package is the schema name of the trigger, and the version identifier is the empty string. Multiple versions of a trigger package are not allowed.

Use the REBIND command to explicitly rebind the trigger package. To specify the name of a trigger package for the bind commands, the name must conform to the rules for an ordinary identifier.

Defining triggers on tables that contain XML columns:

Although a trigger can be defined on a table that contains an XML column, an XML column cannot be referenced with a trigger transition variable in the trigger body. An *SQL-procedure-statement* cannot reference a transition variable that is an XML data type.

Triggers and global variables:

The content of a global variable that is references by a trigger is inherited from the caller. Global variables cannot be modified in or by a trigger.

Restrictions involving pending definition changes:

CREATE TRIGGER is not allowed if the trigger is defined on a table for which there are pending definition changes.

Errors binding triggers:

When a CREATE TRIGGER statement is bound, the SQL statements within the triggered action might not be fully parsed. Syntax errors in those statements might not be caught until the CREATE TRIGGER statement is executed.

Alternative syntax and synonyms:

To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords:

- OLD TABLE as a synonym for OLD_TABLE
- NEW TABLE as a synonym for NEW_TABLE

Examples

Example 1: Create two triggers that track the number of employees that a company manages. The subject table is the EMPLOYEE table, and the triggers increment and decrement a column with the total number of employees in the COMPANY_STATS table. The tables have these columns:

EMPLOYEE table: ID, NAME, ADDRESS, and POSITION
COMPANY_STATS table: NBEMP, NBPRODUCT, and REVENUE

This example shows the use of transition variables in a row trigger to maintain summary data in another table.

Create the first trigger, NEW_HIRE, so that it increments the number of employees each time a new person is hired; that is, each time a new row is inserted into the EMPLOYEE table, increase the value of column NBEMP in table COMPANY_STATS by 1.

```
CREATE TRIGGER NEW_HIRE
   AFTER INSERT ON EMPLOYEE
   FOR EACH ROW MODE DB2SQL
   BEGIN ATOMIC
     UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1;
   END
```

Create the second trigger, FORM_EMP, so that it decrements the number of employees each time an employee leaves the company; that is, each time a row is deleted from the table EMPLOYEE, decrease the value of column NBEMP in table COMPANY_STATS by 1.

```
CREATE TRIGGER FORM_EMP
   AFTER DELETE ON EMPLOYEE
   FOR EACH ROW MODE DB2SQL
   BEGIN ATOMIC
     UPDATE COMPANY_STATS SET NBEMP = NBEMP - 1;
   END
```

Example 2: Create a trigger, REORDER, that invokes user-defined function ISSUE_SHIP_REQUEST to issue a shipping request whenever a parts record is updated and the on-hand quantity for the affected part is less than 10% of its maximum stocked quantity. User-defined function ISSUE_SHIP_REQUEST orders a quantity of the part that is equal to the part's maximum stocked quantity minus its on-hand quantity; the function also ensures that the request is sent to the appropriate supplier.

The parts records are in the PARTS table. Although the table has more columns, the trigger is activated only when columns ON_HAND and MAX_STOCKED are updated.

```
CREATE TRIGGER REORDER
    AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
    REFERENCING NEW AS NROW
    FOR EACH ROW MODE DB2SQL
    WHEN (NROW.ON_HAND < 0.10 * NROW.MAX_STOCKED)
    BEGIN ATOMIC
      VALUES(ISSUE_SHIP_REQUEST(NROW.MAX_STOCKED - NROW.ON_HAND, NROW.PARTNO));
    END
```

Example 3: Repeat the scenario in *Example 2* except use a fullselect instead of a VALUES statement to invoke the user-defined function. This example also shows how to define the trigger as a statement trigger instead of a row trigger. For each row in the transition table that evaluates to true for the WHERE clause, a shipping request is issued for the part.

```
CREATE TRIGGER REORDER
    AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
    REFERENCING NEW_TABLE AS NTABLE
    FOR EACH STATEMENT MODE DB2SQL
      BEGIN ATOMIC
        SELECT ISSUE_SHIP_REQUEST(MAX_STOCKED - ON_HAND, PARTNO)
          FROM NTABLE
        WHERE (ON_HAND < 0.10 * MAX_STOCKED);
    END
```

Example 4: Assume that table EMPLOYEE contains column SALARY. Create a trigger, SAL_ADJ, that prevents an update to an employee's salary that exceeds 20% and signals such an error. Have the error that is returned with an SQLSTATE of '75001' and a description. This example shows that the SIGNAL statement is useful for restricting changes that violate business rules.

```
CREATE TRIGGER SAL_ADJ
  AFTER UPDATE OF SALARY ON EMPLOYEE
  REFERENCING OLD AS OLD_EMP
              NEW AS NEW_EMP
  FOR EACH ROW MODE DB2SQL
  WHEN (NEW_EMP.SALARY > (OLD_EMP.SALARY * 1.20))
    BEGIN ATOMIC
      SIGNAL SQLSTATE '75001' ('Invalid Salary Increase - Exceeds 20
    END
```

Example 5: Assume that the following statements create a table, WEATHER (which stores temperature values in Fahrenheit), and a view, CELSIUS_WEATHER for users who prefer to work in Celsius instead of Fahrenheit:

```
CREATE TABLE WEATHER
   (CITY VARCHAR(25),
    TEMPF DECIMAL(5,2));
CREATE VIEW CELSIUS_WEATHER (CITY, TEMPC) AS
   SELECT CITY, (TEMPF-32)/1.8
   FROM WEATHER;
```

The following INSTEAD OF trigger is used on the CELSIUS_WEATHER view to convert Celsius values to Fahrenheit values and then insert the Fahrenheit value into the WEATHER table:

```
CREATE TRIGGER CW_INSERT INSTEAD OF INSERT
   ON CELSIUS_WEATHER
   REFERENCING NEW AS NEWCW
   FOR EACH ROW MODE DB2SQL
     BEGIN ATOMIC
        INSERT INTO WEATHER VALUES
          (NEWCW.CITY,
           1.8*NEWCW.TEMPC+32)
     END;
```

CREATE TRUSTED CONTEXT

The CREATE TRUSTED CONTEXT statement defines a trusted context at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

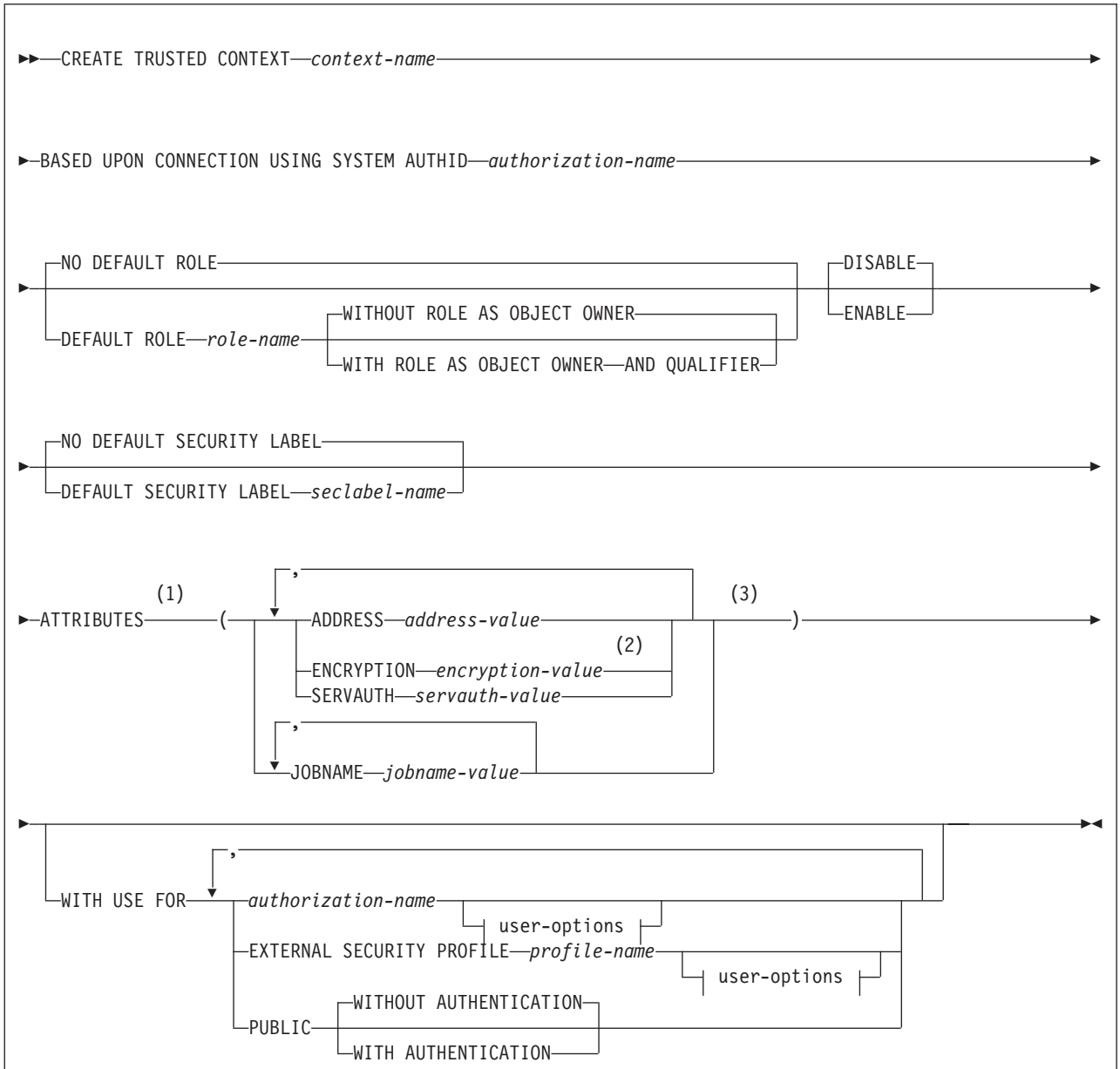
The privilege set that is defined below must include at least one of the following:

- SYSADM authority
- SECADM authority

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the application is bound in a trusted context with the ROLE AS OBJECT OWNER clause specified, a role is the owner. Otherwise, an authorization ID is the owner.

If the statement is dynamically prepared, the privilege set is the privileges that are held by the SQL authorization ID of the process unless the process is within a trusted context and the ROLE AS OBJECT OWNER clause is specified. In that case, the privileges set is the privileges that are held by the role that is associated with the primary authorization ID of the process.

Syntax



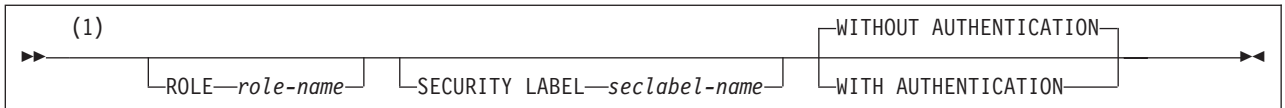
Notes:

1 This clause and the clauses that follow can be specified in any order. Each clause must not be specified more than one time.

2 ENCRYPTION must not be specified more than one time.

3 Each pair of attribute name and corresponding value must be unique.

user-options:



Notes:

1 These clauses can be specified in any order. Each clause must not be specified more than one time.

Description

context-name
Names the trusted context. The name must not identify a trusted context that exists at the current server.

BASED UPON CONNECTION USING SYSTEM AUTHID *authorization-name*
Specifies that the context is a connection that is established by the authorization ID that is specified by *authorization-name*. The system authorization ID is the primary authorization ID. For a remote connection, it is derived from the system user ID that is provided by an external entity, such as a middleware server. For a local connection, the system authorization ID is derived depending on the sources, as specified in Table 130.

Table 130. System authorization ID for a local connection

Source of local connection	System authorization ID
Started task (RRSAF)	USER parameter on JOB statement or RACF USER.
TSO	TSO logon ID
BATCH	USER parameter on JOB statement

authorization-name must not be associated with an existing trusted context.

NO DEFAULT ROLE or DEFAULT ROLE *role-name*
Specifies whether a default role is associated with a trusted connection that is based on the specified trusted context.

NO DEFAULT ROLE
Specifies that the trusted context does not have a default role. The authorization ID of the process is the owner of any object that is created using a trusted connection that is based on this trusted context. That authorization ID must possess all of the privileges that are necessary to create that object.

NO DEFAULT ROLE is the default.

DEFAULT ROLE *role-name*
Specifies that *role-name* is the role for the trusted context. *role-name* must identify a role that exists at the current server. This role is used with the user in a trusted connection that is based on the specified trusted context when the user does not have a user-specified role that is defined as part of the definition of this trusted context.

WITHOUT ROLE AS OBJECT OWNER or WITH ROLE AS OBJECT OWNER AND QUALIFIER
Specifies whether a role is used as the owner of objects that are created using a trusted connection that is based on the specified trusted context.

WITHOUT ROLE AS OBJECT OWNER
Specifies that a role is not used as the owner of the objects that are created

using a trusted connection that is based on the specified trusted context. The authorization ID of the process is the owner of any object that is the created using a trusted connection that is based on this trusted context. That authorization ID must possess all of the privileges that are necessary to create the object.

WITHOUT ROLE AS OBJECT OWNER is the default.

WITH ROLE AS OBJECT OWNER AND QUALIFIER
Specifies that the context assigned role is the owner of the objects that are created using a trusted connection that is based on this trusted context and that role must possess all of the privileges that are necessary to create the object. The context assigned role is the role that is defined for the user within this trusted context, if one is defined. Otherwise, the role is the default role that is associated with the trusted context. The role is also used as the grantor for any GRANT statements that are issued, and the revoker for any REVOKE statement that are issued using a trusted connection that is based on this trusted context.

AND QUALIFIER
Specifies that *role-name* will be used as the default for the CURRENT SCHEMA special register. The *role-name* will also be included in the SQL PATH (in place of CURRENT SQLID).

When **WITH ROLE AS OBJECT OWNER AND QUALIFIER** is not specified, there is no change to the default for the CURRENT SCHEMA special register and the SQL PATH.

DISABLE or ENABLE
Specifies whether the trusted context is created in the enabled or disabled state.

DISABLE
Specified that the trusted context is disabled when it is created. A trusted context that is disabled is not considered when a trusted connection is established. DISABLE is the default.

ENABLE
Specifies that the trusted context is enabled when it is created.

NO DEFAULT SECURITY LABEL or DEFAULT SECURITY LABEL *seclabel-name*
Specifies whether the trusted connection has a default security label.

NO DEFAULT SECURITY LABEL
Specifies that the trusted context does not have a default security label.

DEFAULT SECURITY LABEL *seclabel-name*
Specifies that *seclabel-name* is the default security label for the trusted context and is the security label that is used for multilevel security verification. *seclabel-name* must identify one of the RACF SECLABEL values that is defined for the SYSTEM AUTHID. This security label is used for a trusted connection that is based on the specified trusted context when the user does not have a specific security label defined as part of the definition of this trusted context. In this case, *seclabel-name* must also identify one of the RACF SECLABEL values that is defined for the user.

ATTRIBUTES
Specifies a list of one or more connection trust attributes that are used to define the trusted context.

ADDRESS *address-value*
Specifies the actual communication address that is used by the connection

to communicate with the database manager. The protocol supported is only for TCP/IP. The ADDRESS attribute can be specified multiple times, but each *address-value* must be unique.

When establishing a trusted connection, if multiple values are defined for the ADDRESS attribute for a trusted context, a candidate connection is considered to match this attribute if the address that is used by a connection matches any of the defined values for the ADDRESS attribute of the trusted context.

address-value specifies a string constant that contains the value that is associated with the ADDRESS trust attribute. *address-value* must be an IPv4 address, an IPv6 address, or a secure domain name with a length no greater than 254 bytes. No validation of *address-value* is done at the time the CREATE TRUSTED CONTEXT statement is processed. *address-value* must be left justified within the string constant.

- An IPv4 address is represented as a dotted decimal address. An example of an IPv4 address is 9.112.46.111
- An IPv6 address is represented as a colon hexadecimal address. An example of an IPv6 address is 2001:0DB8:0000:0000:0008:0800:200C:417A. This address can also be express in a compressed form as 2001:DB8::8:800:200C:417A.
- A domain name is converted to an IP address by the domain name server where a resulting IPv4 or IPv6 address is determined. An example of a domain name is www.ibm.com. The gethostbyname socket call is used to resolve the domain name.

ENCRYPTION *encryption-value*
Specifies the minimum level of encryption of the data stream (network encryption).

encryption-value specifies a string constant that contains the value that is associated with the ENCRYPTION trust attribute. *encryption-value* must be left justified within the string constant. ENCRYPTION must not be specified more than one time in the statement. *encryption-value* must be one of the following:

- NONE, which specifies that no specific level of encryption is required.
- LOW, which specifies that a minimum of light encryption is required. LOW corresponds to 64-bit DRDA encryption.
- HIGH, which specifies that strong encryption is required. HIGH corresponds to SSL encryption.

The following table summarizes when a trusted context can be used depending on the encryption that is used by the existing connection. If the trusted context cannot be used for the connection, a warning is returned.

Table 131. Summary of when trusted context can be used by an existing connection

Encryption that is used by the existing connection	Value of the ENCRYPTION clause for the trusted context	Can the trusted context be used for the connection?
No encryption	NONE	Yes
No encryption	LOW	No
No encryption	HIGH	No
Low encryption (64-bit)	NONE	Yes
Low encryption (64-bit)	LOW	Yes
Low encryption (64-bit)	HIGH	No

Table 131. Summary of when trusted context can be used by an existing connection (continued)

Encryption that is used by the existing connection	Value of the ENCRYPTION clause for the trusted context	Can the trusted context be used for the connection?
High encryption (128-bit)	NONE	Yes
High encryption (128-bit)	LOW	Yes
High encryption (128-bit)	HIGH	Yes

JOBNAME *jobname-value*
Specifies the z/OS job name or started task name (depending on the source of the address space) for local applications. The JOBNAME attribute can be specified multiple times, but each *jobname-value* must be unique.

jobname-value specifies a string constant that contains the value that is associated with the JOBNAME trust attribute. *jobname-value* is an EBCDIC 8 byte value that specifies the job name or the started task name. The value must be left justified within the string constant. The last character in the name can be a wildcard character (*) if the first character is an alphabetic character. If the job name ends with a wildcard, any job names that begin with the specified characters are considered for establishing the trusted connection.

The following table lists possible values for the job name depending on the source of the address space.

Table 132. Job name for local connection

Source of the address space	Job name
RRSAF	Job name or started task name
TSO	TSO logon ID
BATCH	Job name on JOB statement

SERVAUTH *servauth-value*
Specifies the name of a resource in the RACF SERVAUTH class. This resource is the network access security zone name that contains the IP address of the connection that is used to communicate with DB2. The SERVAUTH attribute can be specified multiple times but each *servauth-value* must be unique.

servauth-value specifies a string constant that contains the value that is associated with the SERVAUTH trust attribute. *servauth-value* is an EBCDIC 64 byte RACF SERVAUTH CLASS resource name. *servauth-value* must be left justified in the string constant. No validation of *servauth-value* is done at the time the CREATE TRUSTED CONTEXT statement is processed.

WITH USE FOR
Specifies who can use a trusted connection that is based on the specified trusted context.

authorization-name
Specifies that the trusted connection can be used by the specified *authorization-name*. This is the DB2 primary authorization ID. The *authorization-name* must not be specified more than one time in the WITH USE FOR clause.

ROLE *role-name*
Specifies that *role-name* is the role that is used when a trusted

connection is used by the specified *authorization-name*. The *role-name* must identify a role that exists at the current server. The role that is explicitly specified for the user overrides any default role that is associated with the trusted context.

SECURITY LABEL *seclabel-name*
Specifies that *seclabel-name* is the security label to use for multilevel security verification when the trusted connection is used by the specified *authorization-name*. The *seclabel-name* must be one of the RACF SECLABEL values that is defined for the user. The security label that is explicitly specified for the user overrides any default security label that is associated with the trusted context.

WITHOUT AUTHENTICATION or WITH AUTHENTICATION
Specifies whether use of the trusted connection requires authentication of the user.

WITHOUT AUTHENTICATION
Specifies that use of a trusted connection by the user does not require authentication. WITHOUT AUTHENTICATION is the default.

WITH AUTHENTICATION
Specifies that use of a trusted connection requires the authentication token with the authorization ID to authenticate the user. If a trusted connection is established locally, the authentication token is the password that is provided by the CONNECT statement with the USER and USING clauses. If the trusted connection is established from a remote client, the authentication token can be one of the following tokens:

- password
- RACF Passticket
- Kerberos token

EXTERNAL SECURITY PROFILE *profile-name*
Specifies that the trusted connection can be used by the DB2 primary authorization IDs that are permitted to use the specified *profile-name* in RACF. The *profile-name* must not be specified more than one time in the **WITH USE FOR** clause.

ROLE *role-name*
Specifies that *role-name* is the role that is used when a trusted connection is used by any authorization ID permitted to use the specified *profile-name* in RACF. The *role-name* must identify a role that exists at the current server. The role that is explicitly specified for the profile overrides any default role that is associated with the trusted context.

SECURITY LABEL *seclabel-name*
Specifies that *seclabel-name* is the security label to use for multilevel security verification when the trusted connection is used by any authorization ID that is permitted to use the specified *profile-name* in RACF. The *seclabel-name* must be one of the RACF SECLABEL values that is defined for the user. The security label that is explicitly specified for the profile overrides any default security label that is associated with the trusted context.

WITHOUT AUTHENTICATION or WITH AUTHENTICATION
Specifies whether use of the trusted connection requires authentication of the user.

WITHOUT AUTHENTICATION
Specifies that use of a trusted connection by the user does not require authentication. WITHOUT AUTHENTICATION is the default.

WITH AUTHENTICATION
Specifies that use of a trusted connection requires the authentication token with the authorization ID to authenticate the user. If a trusted connection is established locally, the authentication token is the password that is provided by the CONNECT statement with the USER and USING clauses. If the trusted connection is established from a remote client, the authentication token can be one of the following tokens:

- password
- RACF Passticket
- Kerberos token

PUBLIC
Specifies that a trusted connection that is based on the specified trusted context can be used by any user. All users that are using a trusted connection that is defined with PUBLIC use the privileges that are associated with the default role for the associated trusted context. If the default role is not defined for the trusted context, there is no role associated with the users that use a trusted connection that is based on the specified trusted context.

If the default security label for the trusted context is defined, all users that are using the trusted context must have the security label defined as one of the RACF SECLABEL values for the user. The default security label is used for multilevel security verification with all users that are using the trusted context.

WITHOUT AUTHENTICATION or WITH AUTHENTICATION
Specifies whether use of the trusted connection requires authentication of the user.

WITHOUT AUTHENTICATION
Specifies that use of a trusted connection by the user does not require authentication. WITHOUT AUTHENTICATION is the default.

WITH AUTHENTICATION
Specifies that use of a trusted connection requires the authentication token with the authorization ID to authenticate the user. If a trusted connection is established locally, the authentication token is the password that is provided by the CONNECT statement with the USER and USING clauses. If the trusted connection is established from a remote client, the authentication token can be one of the following tokens:

- password
- RACF Passticket
- Kerberos token

Notes

Owner privileges: There are no specific privileges on a trusted context.

Requirement for trusted connections: To use trusted connections, you cannot set the ALL subsystem parameter to ALL and set the RESTART subsystem parameter to DEFER on installation panel DSNTIPS.

Order of precedence for users of a trusted connection: The specifications for a user are determined in the following order of precedence:

- *authorization-name*
- **EXTERNAL SECURITY PROFILE** *profile-name*
- **PUBLIC**

For example, assume that a trusted context is defined with use for JOE WITH AUTHENTICATION, EXTERNAL SECURITY PROFILE SPROFILE WITHOUT AUTHENTICATION, and PUBLIC WITH AUTHENTICATION. Users JOE and SAM are permitted to use the RACF PROFILE SPROFILE. If the trusted connection is used by JOE, authentication is required. If the trusted connection is used by SAM, authentication is not required. However, if user SALLY uses the trusted connection, authentication is required.

User-clause SYSTEM AUTHID considerations: If the *authorization-name* that is specified in the SYSTEM AUTHID clause is the same as the *authorization-name* that is specified in the user-clause *authorization-name*, the role or the security label that is specified for *authorization-name* takes precedence over the default value. The value that is specified for the *profile-name*, is permitted to use the profile. If the authorization name that is specified in the SYSTEM AUTHID clause is permitted to use one of the profile names and is not defined in *authorization-name*, the role or the security label that is specified for that *profile-name* takes precedence over the default value.

If authentication is required for SYSTEM AUTHID, either by specification of the AUTHENTICATION clause in the *user-clause* or by setting the value of the TCP/IP Already Verified subsystem parameter to NO, the authentication requirement takes precedence when establishing a remote trusted connection. For example, if *authorization-name* is the same as the authorization name that is specified for SYSTEM AUTHID and the WITHOUT AUTHENTICATION clause is specified, but the TCP/IP Already Verified subsystem parameter is set to NO, an authentication token is required for SYSTEM AUTHID when the remote trusted connection is established. If *authorization-name* is the SYSTEM AUTHID and the WITH AUTHENTICATION clause is specified, but the TCP/IP Already Verified subsystem parameter is set to YES, an authentication token is still required for SYSTEM AUTHID.

Specifying a role in the definition of a trusted context: The definition of a trusted context can designate a role for a specific authorization ID, and a default role for use for an authorization ID for which a specific role has not been specified in the definition of the trusted context. This role can be used with a trusted connection that is based on the trusted context, but it does not make the role available outside of a trusted connection that is based on the trusted context. When an SQL statement that is not a CREATE, GRANT, or REVOKE statement is issued using a trusted connection, the privileges that are held by a role that is in effect for the authorization ID within the definition of the associated trusted context are considered in addition to other privileges that are directly held by the

authorization ID of the statement. The CREATE, GRANT, and REVOKE statements only consider the privileges of the role that is in effect for the trusted connection, or the authorization ID of the statement if a role is not in effect for the trusted connection. If ROLE AS OBJECT OWNER is in effect for a trusted connection, the role that is in effect for the authorization ID for the trusted connection becomes the owner of any object that is created while using the trusted connection.

When a newly created trusted context takes effect: The newly created trusted context takes effect after the CREATE TRUSTED CONTEXT statement is committed. If the CREATE TRUSTED CONTEXT statement results in an error or is rolled back, no trusted context is created.

Examples

Example 1: The following statement creates a trusted context called CTX1, which is based on a connection and can only be used by users JOE and SAM. Authentication information is required for JOE to use the trusted connection. The trusted context specifies a default role called CTXROLE. However, when JOE uses the trusted connection, the default role is overridden by the user role, ROLE1. When SAM uses the trusted connection, SAM uses the default role. CTX1 is enabled when it is created.

```
CREATE TRUSTED CONTEXT CTX1
    BASED UPON CONNECTION USING SYSTEM AUTHID ADMF001
    ATTRIBUTES (ADDRESS '9.30.131.203',
                ENCRYPTION 'LOW')
    DEFAULT ROLE CTXROLE
    ENABLE
    WITH USE FOR SAM, JOE ROLE ROLE1 WITH AUTHENTICATION;
```

Example 2: The following statement creates a trusted context, CTX2, for a started task, WASPROD. CTX2 is based on a connection, can be used by user SALLY, specifies a default role CTXROLE, and is enabled when it is created. SALLY uses the default role that is associated with the trusted context.

```
CREATE TRUSTED CONTEXT CTX2
     BASED UPON CONNECTION USING SYSTEM AUTHID ADMF002
     ATTRIBUTES (JOBNAME 'WASPROD')
     DEFAULT ROLE CTXROLE WITH ROLE AS OBJECT OWNER AND QUALIFIER
     ENABLE
     WITH USE FOR SALLY;
```

CREATE TYPE

The CREATE TYPE statement defines a user-defined data type at the current server.

The following types of user-defined data types can be defined:

Array A user-defined data type that is an ordinary array or an associative array. The elements of an array type are based on one of the built-in data types. See "CREATE TYPE (array)" on page 1511.

Distinct
A user-defined data type that shares a common representation with one of the built-in data types. Functions that cast between the user-defined distinct type and the source built-in data type are generated when the user-defined distinct type is created. Optionally, support for comparison operations to use with the user-defined distinct type can be generated when the user-defined distinct type is created. See "CREATE TYPE (distinct)" on page 1516.

CREATE TYPE (array)

The CREATE TYPE (array) statement defines an array type at the current server. An array type is a user-defined data type that is an ordinary array or an associative array. The elements of an array type are based on one of the built-in data types.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package.

Authorization

The privilege set that is defined below must include at least one of the following:

- The CREATEIN privilege on the schema
- SYSADM or SYSCTRL authority
- System DBADM

The authorization ID that matches the schema name implicitly has the CREATEIN privilege on the schema.

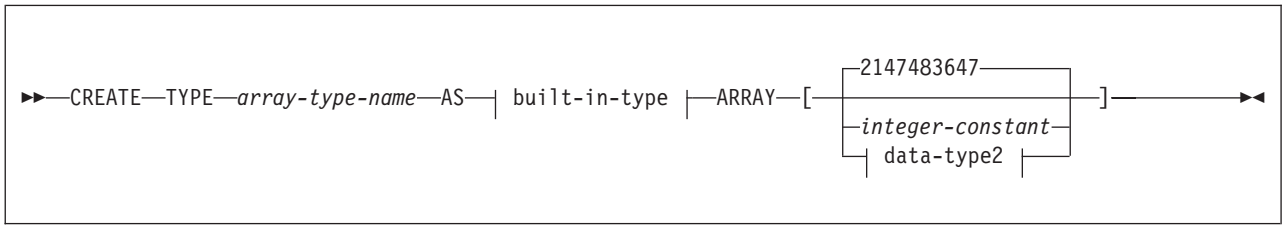
Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package.

If the statement is running under a trusted context for which the ROLE AS OBJECT OWNER clause is specified, the owner is a role. The implicit schema match does not apply, and this role needs to include one of the previously listed conditions.

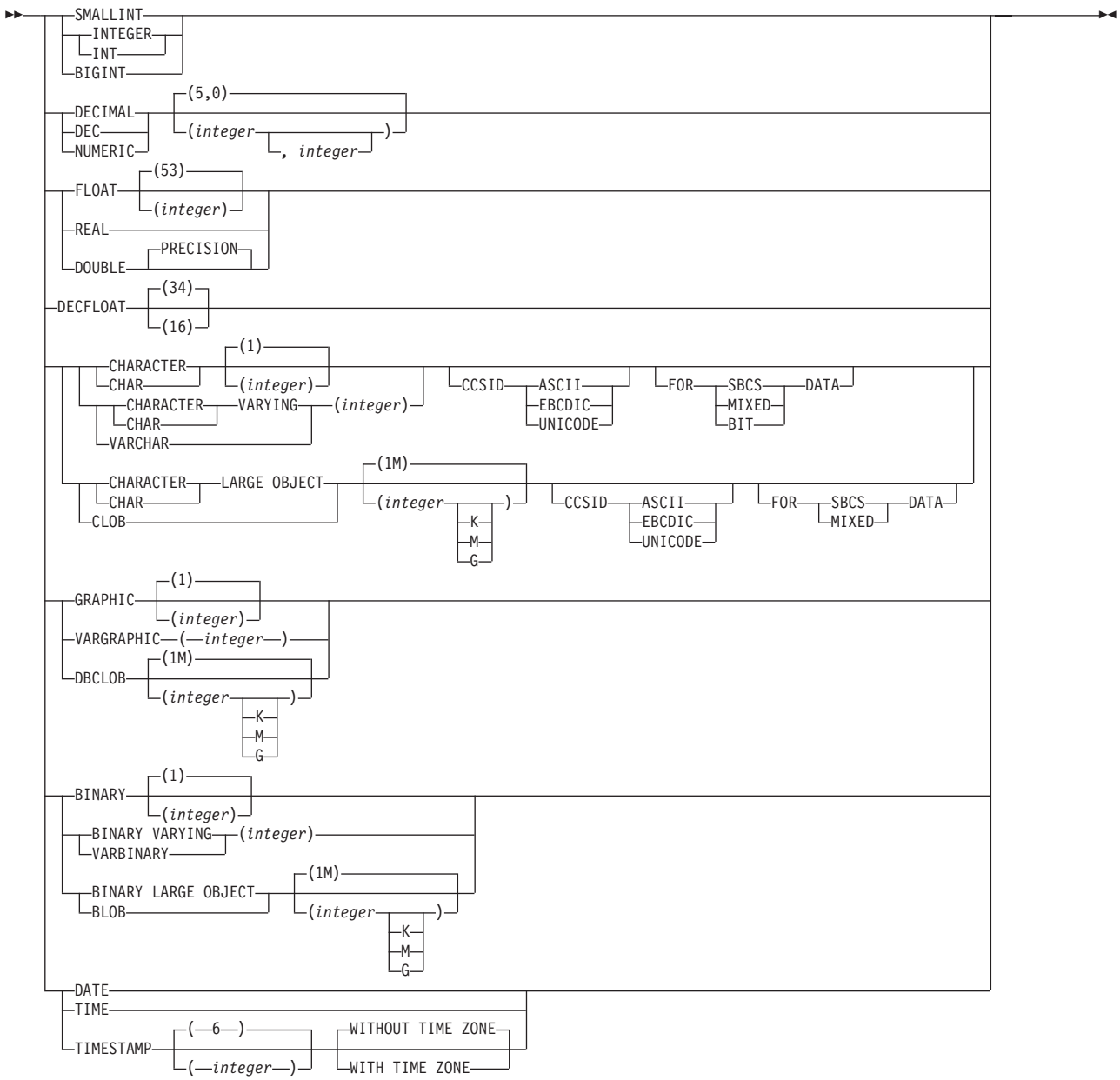
If the statement is dynamically prepared and is not running in a trusted context for which the ROLE AS OBJECT OWNER clause is specified, the privilege set is the set of privileges that are held by the SQL authorization ID of the process. The specified distinct type name can include a schema name (a qualifier). If the schema name is not the same as the SQL authorization ID of the process, one of the following conditions must be met:

- The privilege set includes SYSADM or SYSCTRL authority.
- The SQL authorization ID of the process has the CREATEIN privilege on the schema.

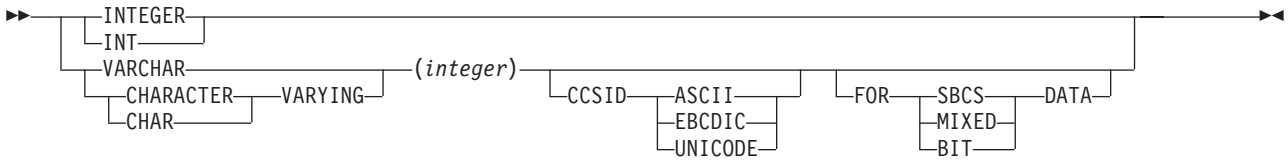
Syntax



built-in-type:



data-type2:



Description

array-type-name
Names the array type. The name, including the implicit or explicit qualifier, must not identify any other built-in or user-defined type that exists at the current server.

The unqualified form of *array-type-name* must not be any of the following system-reserved keywords, even if you specify them as delimited identifiers:

ALL	LIKE	UNIQUE
AND	MATCH	UNKNOWN
ANY	NOT	=
BETWEEN	NULL	¬=
DISTINCT	ONLY	<
EXCEPT	OR	<=
EXISTS	OVERLAPS	¬<
FALSE	SIMILAR	>
FOR	SOME	>=
FROM	TABLE	¬>
IN	TRUE	<>
IS	TYPE	

The schema name can be 'SYSTOOLS' if the user who executes the CREATE statement has SYSADM or SYSCTRL privilege. Otherwise, the schema name must not begin with 'SYS' unless the schema name is 'SYSADM'.

built-in-type
Specifies the built-in data type of the array elements. The data type must not be ROWID or XML. For more information on built-in data types, see built-in-type in CREATE TABLE.

CCSID ASCII, EBCDIC, or UNICODE in a *built-in-type* specification
If the data type is a character string, and a CCSID clause is not specified for *built-in-type*, the default CCSID is determined as follows:

- If *data-type2* is a character string data type with an explicit CCSID clause, that same CCSID value is used for *built-in-type*.
- If *data-type2* is a character string data type without an explicit CCSID clause, the CCSID for *built-in-type* is determined from the encoding scheme that is indicated by the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

If a CCSID clause is specified for *built-in-type* and for *data-type2*, the CCSID values must be the same.

FOR SBCS, MIXED, or BIT DATA in a *built-in-type* specification
Specifies a subtype for a character string data type (VARCHAR). Do not use this clause with any other data type.

SBCS Single-byte data.

MIXED
Mixed data. Do not specify MIXED if the value of field MIXED DATA on installation panel DSNTIPF is NO unless the CCSID UNICODE clause is also specified.

BIT Bit data.

If you do not specify the FOR SBCS DATA, FOR MIXED DATA, or FOR BIT DATA clause, the default value is determined as follows:

- For ASCII or EBCDIC data:
  - The default is SBCS when the value of field MIXED DATA on installation panel DSNTIPF is NO.
  - The default is MIXED when the value is YES.
- For Unicode data, the default subtype is MIXED.

ARRAY[*integer-constant*]
Specifies that the type is an ordinary array with a maximum cardinality of *integer-constant*. The value must be an integer that is greater than 0 and less

than or equal to the largest positive integer value (2147483647). The default is 2147483647. Each varying-length string array element is allocated as its maximum length.

The cardinality of an array value is determined by the highest element position that is assigned to the array value. The maximum cardinality of an array is limited by the total amount of memory that is available to DB2 applications. Therefore, although an array with a large cardinality can be created, not all elements might be available for use. An attempt to assign a value to an array element when there is not enough memory results in an error.

ARRAY[*data-type2***]**
Specifies that the type is an associative array that is indexed by values of data type *data-type2*. The data type must be the INTEGER or VARCHAR data type. The value that is specified as the index during assignment of a value to an array element must be assignable to a value of *data-type2*.

The cardinality of an array value is determined by the number of unique index values that are used when during assignment of array elements.

CCSID ASCII, EBCDIC, or UNICODE in a *data-type2* specification
If the data type is a character string, and a CCSID clause is not specified for *data-type2*, the default CCSID is determined as follows:

- If *built-in-type* is a character string data type with an explicit CCSID clause, that same CCSID value is used for *data-type2*.
- If *built-in-type* is a character string data type without an explicit CCSID clause, the CCSID for *data-type2* is determined from the encoding scheme that is indicated by the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

If a CCSID clause is specified for *built-in-type* and for *data-type2*, the CCSID values must be the same.

FOR SBCS, MIXED, or BIT DATA in a *data-type2* specification
Specifies a subtype for a character string data type (VARCHAR). Do not use this clause with any other data type.

SBCS Single-byte data.

MIXED
Mixed data. Do not specify MIXED if the value of field MIXED DATA on installation panel DSNTIPF is NO unless the CCSID UNICODE clause is also specified.

BIT Bit data.

If you do not specify the FOR SBCS DATA, FOR MIXED DATA, or FOR BIT DATA clause, the default value is determined as follows:

- For ASCII or EBCDIC data:
  - The default is SBCS when the value of field MIXED DATA on installation panel DSNTIPF is NO.
  - The default is MIXED when the value is YES.
- For Unicode data, the default subtype is MIXED.

Notes

Array type usage: A user-defined array type can only be used as the data type of:

- An SQL variable
- A parameter or RETURNS *data-type* of an SQL scalar function

- A parameter of a native SQL procedure
- The target data type for a CAST specification

Generated cast functions: The successful execution of the CREATE TYPE (array) statement causes the DB2 database manager to generate cast functions for the user-defined array type. Those cast functions are recorded in the DB2 catalog. The unqualified names of the two cast functions are ARRAY and the name of the array type. A generated cast function cannot be explicitly dropped. The cast functions that are generated for an array type are implicitly dropped when the array type is dropped with the DROP statement.

Examples

Example 1: Create an ordinary array user-defined type named PHONENUMBERS, with a maximum of 50 elements. The elements are of the DECIMAL(10,0) data type.

```
CREATE TYPE PHONENUMBERS AS DECIMAL(10,0) ARRAY[50];
```

Example 2: Create an ordinary array user-defined type named NUMBERS, in the schema GENERIC. You do not know the maximum number of elements, so you use the default value. The elements are of the DECFLOAT(34) data type.

```
CREATE TYPE GENERIC.NUMBERS AS DECFLOAT(34) ARRAY[];
```

Example 3: Create an associative array user-defined type named PERSONAL_PHONENUMBERS. The elements are of the DECIMAL(16, 0) data type. The array type is indexed by strings such as 'Home', 'Work', or 'Cell', so the index data type must be VARCHAR.

```
CREATE TYPE PERSONAL_PHONENUMBERS AS DECIMAL(16,0) ARRAY[VARCHAR(8)];
```

Example 4: Create an associative array user-defined type named CAPITALSARRAY. The elements are capital cities. The index values are province, territory, or country names, so the index data type must be VARCHAR.

```
CREATE TYPE CAPITALSARRAY AS VARCHAR(30) ARRAY[VARCHAR(20)];
```

Example 5: Create an associative array user-defined type named PRODUCTS. The elements are product descriptions of up to 40 characters. The index values are product numbers, which have the INTEGER data type.

```
CREATE TYPE PRODUCTS AS VARCHAR(40) ARRAY[INTEGER];
```

CREATE TYPE (distinct)

The CREATE TYPE (distinct) statement defines a distinct type, which is a data type that a user defines. A distinct type must be based on one of the built-in data types.

Successful execution of the statement also generates:

- A function to cast between the distinct type and its source type
- A function to cast between the source type and its distinct type
- As appropriate, support for the use of comparison operators with the distinct type

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set that is defined below must include at least one of the following:

- The CREATEIN privilege on the schema
- SYSADM or SYSCTRL authority
- System DBADM

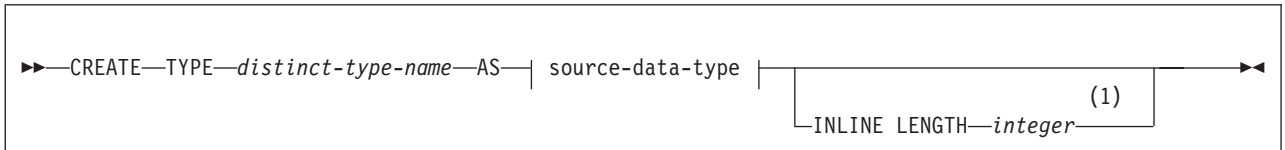
The authorization ID that matches the schema name implicitly has the CREATEIN privilege on the schema.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the owner is a role, the implicit schema match does not apply and this role needs to include one of the previously listed conditions.

If the statement is dynamically prepared and is not running in a trusted context for which the ROLE AS OBJECT OWNER clause is specified, the privilege set is the set of privileges that are held by the SQL authorization ID of the process. The specified distinct type name can include a schema name (a qualifier). If the schema name is not the same as the SQL authorization ID of the process, one of the following conditions must be met:

- The privilege set includes SYSADM or SYSCTRL authority.
- The SQL authorization ID of the process has the CREATEIN privilege on the schema.

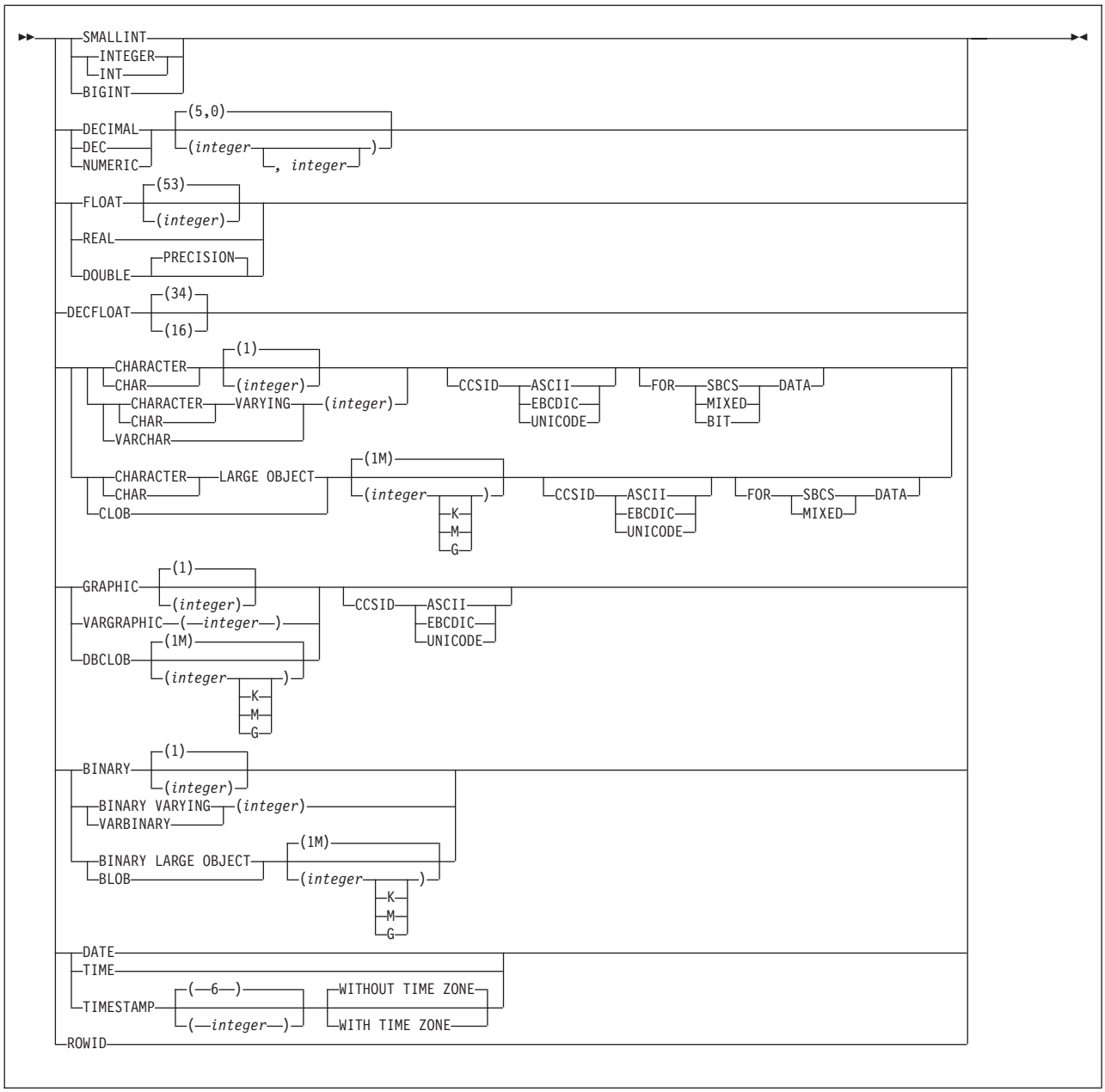
Syntax



Notes:

1 INLINE LENGTH can only be specified when *source-data-type* is a LOB data type.

source-data-type



Description

distinct-type-name

Names the distinct type. The name, including the implicit or explicit qualifier, must not identify a distinct type that exists at the current server.

- The unqualified form of *distinct-type-name* must not be the name of a built-in data type, BOOLEAN, or any of following system-reserved keywords even if you specify them as delimited identifiers:

```
ALL                 LIKE                UNIQUE
AND                 MATCH               UNKNOWN
ANY                 NOT                 =
BETWEEN             NULL                ¬=
DISTINCT            ONLY                <
EXCEPT              OR                  <=
EXISTS              OVERLAPS            ¬<
FALSE               SIMILAR             >
FOR                 SOME                >=
FROM                TABLE               ¬>
IN                  TRUE                <>
IS                  TYPE
```

- The qualified form of *distinct-type-name* is an SQL identifier (the schema name) followed by a period and an SQL identifier.

The schema name can be 'SYSTOOLS' if the user who executes the CREATE statement has SYSADM or SYSCTRL privilege. Otherwise, the schema name must not begin with 'SYS' unless the schema name is 'SYSADM'.

source-data-type

Specifies the data type that is used as the basis for the internal representation of the distinct type. The data type must be a built-in data type. For more information on built-in data types, see built-in-type.

If the distinct type is based on a character or graphic string data type, the FOR clause indicates the subtype. If you do not specify the FOR clause, the distinct type is defined with the default subtype. For ASCII or EBCDIC data, the default is SBCS when the value of field MIXED DATA on installation panel DSNTIPF is NO. The default is MIXED when the value is YES. For UNICODE character data, the default subtype is mixed.

If the distinct type is based on a string data type, the CCSID clause indicates whether the encoding scheme of the data is ASCII, EBCDIC or UNICODE. If you do not specify CCSID ASCII, CCSID EBCDIC, or UNICODE, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

INLINE LENGTH *integer*

Specifies the default inline length for columns that reference the distinct type. INLINE LENGTH can only be specified when *source-data-type* is a LOB data type. Only columns in a table that is in a universal table space can inherit the specified inline length for the distinct type. If the table is not in a universal table space, the specified inline length is ignored.

Where *source-data-type* is BLOB and CLOB, *integer* specifies the maximum number of bytes that are stored in the base table space for columns that reference this distinct type. *integer* must be between 0 and 32680 (inclusive) for a BLOB or CLOB *source-data-type*.

Where *source-data-type* is DBCLOB, *integer* specifies the maximum number of double-byte characters that are stored in the table space for columns that reference the distinct type. *integer* must be between 0 and 16340 (inclusive) for a DBCLOB *source-data-type*.

If INLINE LENGTH is specified with a value of 0 for *integer*, any column that references the distinct type will not have an inline length unless the CREATE TABLE or ALTER TABLE ADD statement specifies an inline length for the column.

If INLINE LENGTH is not specified, any column that reference the distinct type takes its default vale from the value of the LOB INLINE LENGTH parameter on installation panel DSNTIPD.

integer cannot be greater than the maximum length of the distinct type.

Notes

Owner privileges:

The owner of the distinct type is authorized to define columns, parameters, or variables with the distinct type (USAGE privilege) with the ability to grant these privileges to others. See "GRANT (type or JAR file privileges)" on page 1725. The owner is also authorized to invoke the generated cast function (EXECUTE privilege; see "GRANT (function or procedure privileges)" on page 1703). The owner is given the USAGE and EXECUTE privileges with the GRANT option. For more information about ownership of the object, see "Authorization, privileges, permissions, masks, and object ownership" on page 70.

Source data types with DBCS or mixed data:

When the implicit or explicit encoding scheme is ASCII or EBCDIC and the source data type is graphic or a character type is MIXED DATA, then the value of field FOR MIXED DATA on installation panel DSNTIPF must be YES; otherwise, an error occurs.

Generated cast functions:

The successful execution of the CREATE TYPE (distinct) statement causes DB2 to generate the following cast functions:

- A function to convert from the distinct type to its source data type
- A function to convert from the source data type to the distinct type
- A function to cast from a data type *A* to distinct type *DT*, where *A* is promotable to the source data type *S* of distinct type *DT*

  For some source data types, DB2 supports an additional function to convert from:

  – INTEGER to the distinct type if the source type is SMALLINT
  – VARCHAR to the distinct type if the source type is CHAR
  – VARGRAPHIC to the distinct type if the source type is GRAPHIC
  – VARBINARY to the distinct type if the source type is BINARY
  – DOUBLE to the distinct type if the source type is REAL

The cast functions are created as if the following statements were executed:

```
CREATE FUNCTION source-type-name (distinct-type-name)
   RETURNS source-type-name ...
CREATE FUNCTION distinct-type-name (source-type-name)
   RETURNS distinct-type-name ...
```

Even if you specified a length, precision, or scale for the source data type in the CREATE TYPE (distinct) statement, the name of the cast function that converts from the distinct type to the source type is simply the name of the source data type. The data type of the value that the cast function returns includes any length, precision, or scale values that you specified for the source data type. (See Table 133 on page 1520 for details.)

The name of the cast function that converts from the source type to the distinct type is the name of the distinct type. The input parameter of the cast function has the same data type as the source data type, including the length, precision, and scale.

For example, assume that a distinct type named T_SHOESIZE is created with the following statement:

```
CREATE TYPE (distinct) CLAIRE.T_SHOESIZE AS VARCHAR(2)
```

When the statement is executed, DB2 also generates the following cast functions. VARCHAR converts from the distinct type to the source type, and T_SHOESIZE converts from the source type to the distinct type.

```
FUNCTION CLAIRE.VARCHAR (CLAIRE.T_SHOESIZE) RETURNS SYSIBM.VARCHAR (2)
FUNCTION CLAIRE.T_SHOESIZE (SYSIBM.VARCHAR (2)) RETURNS CLAIRE.T_SHOESIZE
```

Notice that function VARCHAR returns a value with a data type of VARCHAR(2) and that function T_SHOESIZE has an input parameter with a data type of VARCHAR(2).

The schema of the generated cast functions is the same as the schema of the distinct type. No other function with the same name and function signature must already exist in the database.

In the preceding example, if T_SHOESIZE had been sourced on a SMALLINT, CHAR, or GRAPHIC data type instead of a VARCHAR data type, another cast function would have been generated in addition to the two functions to cast between the distinct type and the source data type. For example, assume that T_SHOESIZE is created with this statement:

```
CREATE TYPE (distinct) CLAIRE.T_SHOESIZE AS CHAR(2)
```

When the statement is executed, DB2 generates these cast functions:

```
FUNCTION CLAIRE.CHAR (CLAIRE.T_SHOESIZE) RETURNS SYSIBM.CHAR (2)
FUNCTION CLAIRE.T_SHOESIZE (SYSIBM.CHAR (2)) RETURNS CLAIRE.T_SHOESIZE
FUNCTION CLAIRE.T_SHOESIZE (SYSIBM.VARCHAR (2)) RETURNS CLAIRE.T_SHOESIZE
```

Notice that the third function enables the casting of a VARCHAR(2) to T_SHOESIZE. This additional function is created to enable casting a constant, such as 'AB', directly to the distinct type. Without the additional function, you would have to first cast 'AB', which has a data type of VARCHAR, to a data type of CHAR and then cast it to the distinct type.

You cannot explicitly drop a generated cast function. The cast functions that are generated for a distinct type are implicitly dropped when the distinct type is dropped with the DROP statement.

For each built-in data type that can be the source data type for a distinct type, the following table gives the names of the generated cast functions, the data types of the input parameters, and the data types of the values that the functions returns.

Table 133. CAST functions on distinct types

Source type name	Function name	Parameter-type	Return-type
SMALLINT	*distinct-type-name*	SMALLINT	*distinct-type-name*
	distinct-type-name	INTEGER	*distinct-type-name*
	SMALLINT	*distinct-type-name*	SMALLINT
INTEGER	*distinct-type-name*	INTEGER	*distinct-type-name*
	INTEGER	*distinct-type-name*	INTEGER

Table 133. CAST functions on distinct types (continued)

Source type name	Function name	Parameter-type	Return-type
BIGINT	*distinct-type-name*	BIGINT	*distinct-type-name*
	BIGINT	*distinct-type-name*	BIGINT
DECIMAL	*distinct-type-name*	DECIMAL (p,s)	*distinct-type-name*
	DECIMAL	*distinct-type-name*	DECIMAL (p,s)
NUMERIC	*distinct-type-name*	DECIMAL (p,s)	*distinct-type-name*
	DECIMAL	*distinct-type-name*	DECIMAL (p,s)
REAL	*distinct-type-name*	REAL	*distinct-type-name*
	distinct-type-name	DOUBLE	*distinct-type-name*
	REAL	*distinct-type-name*	REAL
DECFLOAT	*distinct-type-name*	DECFLOAT(n)	DECFLOAT(n)
	DECFLOAT	*distinct-type-name*	DECFLOAT(n)
FLOAT(*n*) where *n*<=21	*distinct-type-name*	REAL	*distinct-type-name*
	distinct-type-name	DOUBLE	*distinct-type-name*
	REAL	*distinct-type-name*	REAL
FLOAT(*n*) where *n*>21	*distinct-type-name*	DOUBLE	*distinct-type-name*
	DOUBLE	*distinct-type-name*	DOUBLE
FLOAT	*distinct-type-name*	DOUBLE	*distinct-type-name*
	DOUBLE	*distinct-type-name*	DOUBLE
DOUBLE	*distinct-type-name*	DOUBLE	*distinct-type-name*
	DOUBLE	*distinct-type-name*	DOUBLE
DOUBLE PRECISION	*distinct-type-name*	DOUBLE	*distinct-type-name*
	distinct-type-name	CHAR (n)	*distinct-type-name*
	CHAR	*distinct-type-name*	CHAR (n)
	distinct-type-name	VARCHAR (n)	*distinct-type-name*
	DOUBLE	*distinct-type-name*	DOUBLE
CHAR CHARACTER	*distinct-type-name*	CHAR (n)	*distinct-type-name*
	CHAR	*distinct-type-name*	CHAR (n)
	distinct-type-name	VARCHAR (n)	*distinct-type-name*
VARCHAR CHARACTER VARYING CHAR VARYING	*distinct-type-name*	VARCHAR (n)	*distinct-type-name*
	VARCHAR	*distinct-type-name*	VARCHAR (n)
CLOB	*distinct-type-name*	CLOB (n)	*distinct-type-name*
	CLOB	*distinct-type-name*	CLOB (n)
GRAPHIC	*distinct-type-name*	GRAPHIC (n)	*distinct-type-name*
	GRAPHIC	*distinct-type-name*	GRAPHIC (n)
	distinct-type-name	VARGRAPHIC (n)	*distinct-type-name*
VARGRAPHIC	*distinct-type-name*	VARGRAPHIC (n)	*distinct-type-name*
	VARGRAPHIC	*distinct-type-name*	VARGRAPHIC (n)
DBCLOB	*distinct-type-name*	DBCLOB (n)	*distinct-type-name*
	DBCLOB	*distinct-type-name*	DBCLOB (n)

Table 133. CAST functions on distinct types (continued)

Source type name	Function name	Parameter-type	Return-type
BINARY	*distinct-type-name*	BINARY(n)	*distinct-type-name*
	BINARY	*distinct-type-name*	BINARY(n)
	distinct-type-name	VARBINARY(n)	*distinct-type-name*
VARBINARY	*distinct-type-name*	VARBINARY(n)	*distinct-type-name*
	VARBINARY	*distinct-type-name*	VARBINARY(n)
BLOB	*distinct-type-name*	BLOB (n)	*distinct-type-name*
	BLOB	*distinct-type-name*	BLOB (n)
DATE	*distinct-type-name*	DATE	*distinct-type-name*
	DATE	*distinct-type-name*	DATE
TIME	*distinct-type-name*	TIME	*distinct-type-name*
	TIME	*distinct-type-name*	TIME
TIMESTAMP	*distinct-type-name*	TIMESTAMP	*distinct-type-name*
	TIMESTAMP	*distinct-type-name*	TIMESTAMP(*p*) WITHOUT TIME ZONE
TIMESTAMP(*p*) WITH TIME ZONE	*distinct-type-name*	TIMESTAMP WITH TIME ZONE	*distinct-type-name*
	TIMESTAMP_TZ	*distinct-type-name*	TIMESTAMP(*p*) WITH TIME ZONE
ROWID	*distinct-type-name*	ROWID	*distinct-type-name*
	ROWID	*distinct-type-name*	ROWID

Notes: NUMERIC and FLOAT are not recommended when creating a distinct type for a portable application. Use DECIMAL and DOUBLE (or REAL) instead.

Built-in functions:

When a distinct type is defined, the built-in functions (such as AVG, MAX, and LENGTH) are not automatically supported for the distinct type. You can use a built-in function on a distinct type only after a sourced user-defined function, which is based on the built-in function, has been created for the distinct type. For information on defining sourced user-defined functions, see "CREATE FUNCTION (sourced)" on page 1210.

Arithmetic operators with distinct type operands:

A distinct type cannot be used with arithmetic operators even if its source data type is numeric.

For additional information see "Arithmetic with distinct type operands" on page 250.

Alternative syntax and synonyms:

The WITH COMPARISONS clause, which specifies that system-generated comparison operators are to be created for comparing two instances of the distinct type, can be specified as the last clause of the statement. Use WITH COMPARISONS only if it is required for compatibility with other products in the DB2 family. If the source data type is either BLOB, CLOB, or DBCLOB and WITH COMPARISONS is specified, a warning occurs as in previous releases.

To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following clauses:

- DISTINCT TYPE as a synonym for TYPE
- TIMEZONE can be specified as an alternative to TIME ZONE

.

Examples

Example 1: Create a distinct type named SHOESIZE that is based on an INTEGER data type.

```
CREATE TYPE SHOESIZE AS INTEGER;
```

The successful execution of this statement also generates two cast functions. Function INTEGER(SHOESIZE) returns a value with data type INTEGER, and function SHOESIZE(INTEGER) returns a value with distinct type SHOESIZE.

Example 2: Create a distinct type named MILES that is based on a DOUBLE data type.

```
CREATE TYPE MILES AS DOUBLE;
```

The successful execution of this statement also generates two cast functions. Function DOUBLE(MILES) returns a value with data type DOUBLE, and function MILES(DOUBLE) returns a value with distinct type MILES.

CREATE VARIABLE

The CREATE VARIABLE statement creates a global variable at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set that is defined below must include at least one of the following:

- The CREATEIN privilege on the schema
- System DBADM authority
- SYSADM authority
- SYSCTRL authority

Privilege set: The authorization ID that matches the schema name implicitly has the CREATEIN privilege on the schema. If the statement is embedded in an application program, the privilege set is the set of privileges that are held by the owner of the package. If the owner is a role, the implicit schema does not apply and this role needs to include one of the previously listed privileges or authorities.

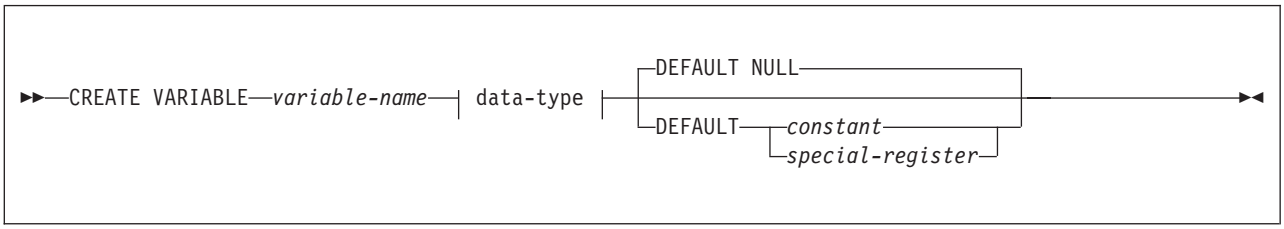
If the statement is dynamically prepared, the privilege set is the privileges that are held by the SQL authorization ID of the process unless the process is within a trusted context and the ROLE AS OBJECT OWNER AND QUALIFIER clause is in effect. If the schema name is not the same as the SQL authorization ID of the process, one of the following conditions must be met:

- The privilege set includes SYSADM or SYSCTRL authority.
- The SQL authorization ID of the process has the CREATEIN privilege on the schema.

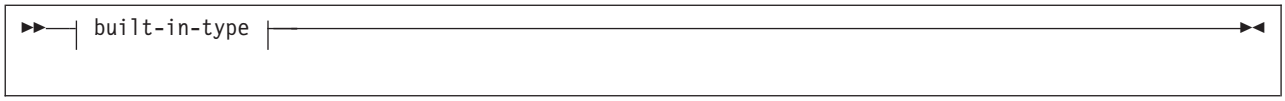
When the ROLE AS OBJECT OWNER AND QUALIFIER clause is in effect, the privilege set is the set privileges that are held by the role. If the schema name does not match this role, one of the following conditions must be met:

- The privilege set includes SYSADM or SYSCTRL authority.
- This role has the CREATEIN privilege on the schema.

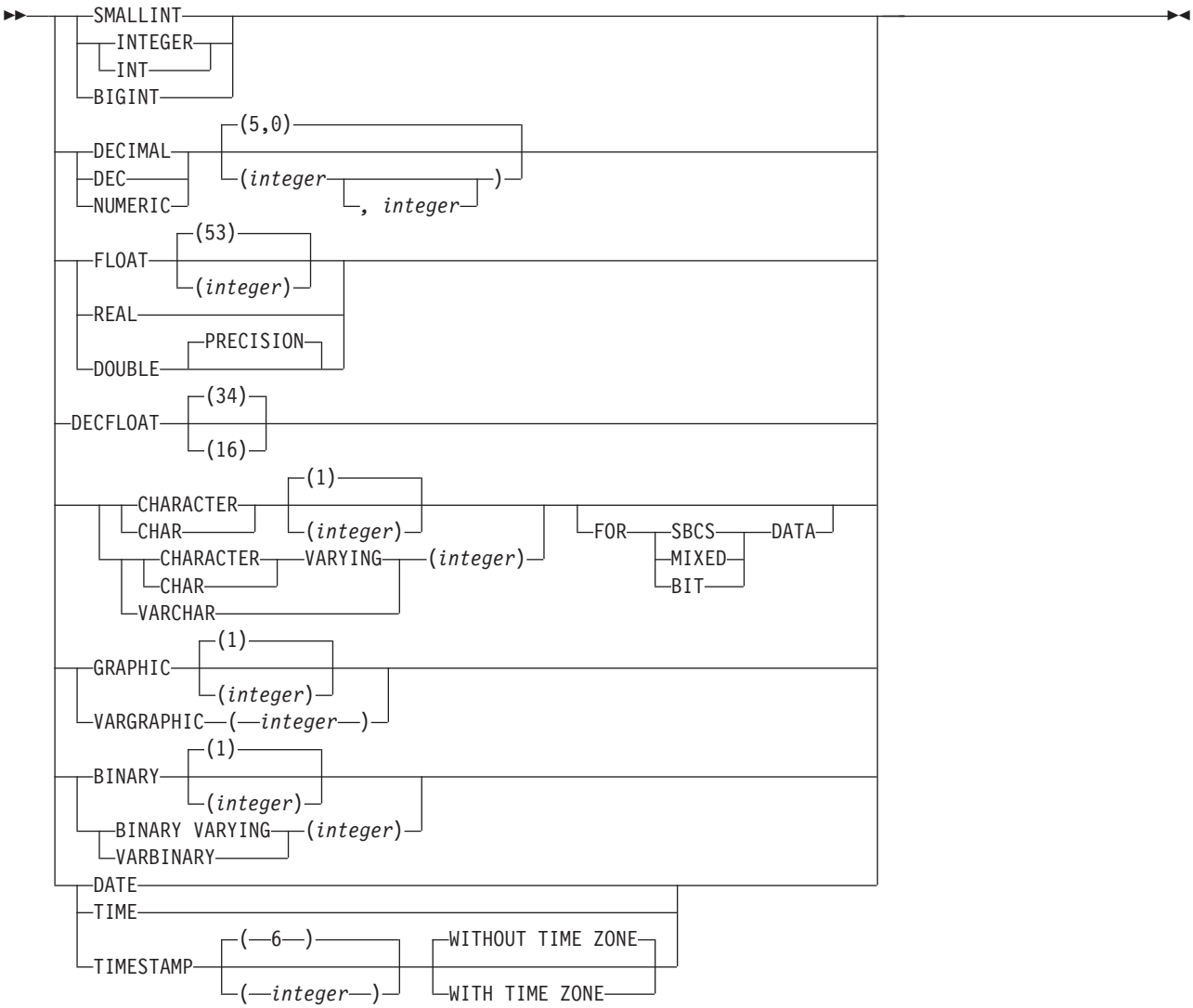
Syntax



data-type:



built-in-type:



Description

variable name
Names the global variable. The name, including the implicit or explicit qualifier, must not identify a global variable that exists at the current server. If the qualifier is not specified, the contents of the CURRENT SCHEMA special register is used.

data-type
Specifies the data type of the global variable.

built-in-type
The data type of the global variable is a built-in type. For information about the data types, see built-in-type. The data type cannot be a LOB data type, ROWID, or XML.

DEFAULT
Specifies a value for the global variable when it is first referenced in the session. The default value is determined on this first reference. If the DEFAULT clause is not specified, the default for the global variable is the null value.

NULL
Specifies null as the default value for the global variable. The value of a global variable is always nullable.

constant
Specifies that the value of the constant is the default value for the global variable. The value of the constant must conform to the rules for assigning that value to the global variable. *constant* cannot be any of the constants NAN, SNAN, or INFINITY.

special-register
Specifies that the value of the special register, when the global variable is instantiated, is used as the default value of the global variable. The value of the specified special register must conform to the rules for assigning that value to the global variable. The following special registers must not be specified:
- CURRENT GET_ACCEL_ARCHIVE
- CURRENT QUERY_ACCELERATION
- CURRENT TEMPORAL BUSINESS_TIME
- CURRENT TEMPORAL SYSTEM_TIME

Notes

Session scope:
global variables have a session scope. Although they are available for use to all sessions that are active at the current server, the value of the global variable is private for each session.

Modifications to the value of a global variable:
global variables are not under transaction control. Modifications to the value of a global variable are not preserved when the transaction ends with either a COMMIT or ROLLBACK statement.

Privileges to use a global variable:
Reading from or writing to a global variable requires that the authorization ID or role that is in effect have the appropriate privileges on the global variable. The owner of the variable is implicitly granted all privileges on the variable.

Setting the default value:
After a global variable has been created, it is instantiated to its default value when it is first referenced within a given scope. If a global variable is referenced within a statement, it is instantiated independently from the execution of that statement.

Using a newly created global variable:
If a global variable is created within a session, it cannot be used by other sessions until the unit of work has committed. However, the newly created global variable can be used within the session in which it is created before the unit of work commits.

CREATE VIEW

The CREATE VIEW statement creates a view on tables or views at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

For every table or view identified in the *fullselect*, the privilege set that is defined below must include at least one of the following:

- The SELECT privilege on the table or view
- Ownership of the table or view
- DBADM authority for the database (tables only)
- DATAACCESS authority
- SYSADM authority
- SQLADM authority (catalog tables only)
- System DBADM authority (catalog tables only)
- ACCESSCTRL authority (catalog tables only)
- SYSCTRL authority (catalog tables only)

If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

Authority requirements depend in part on the choice of the owner of the view. For information on how to choose the owner, see the description of *view-name* in "Description" on page 1109.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the of the owner of the plan or package. If the application is bound in a trusted context with the ROLE AS OBJECT OWNER clause specified, a role is the owner. Otherwise, an authorization ID is the owner.

- If this privilege set includes SYSADM authority, the owner of the view can be any authorization ID. If that set includes SYSCTRL but not SYSADM authority, the following is true: the owner of the view can be any authorization ID, provided the view does not refer to user tables or views in the first FROM clause of its defining fullselect. (It could refer instead, for example, to catalog tables or views thereof.)

  If the view satisfies the rules in the preceding paragraph, and if no errors are present in the CREATE statement, the view is created, even if the owner has no privileges at all on the tables and views identified in the fullselect of the view definition.
- If the privilege set includes system DBADM authority, the owner of the view can be any authorization ID. However, to create a view on a user table, either the owner of the view or the creator must have the SELECT privilege on all the tables or views in the CREATE VIEW statement.

- If the privilege set lacks system DBADM, SYSADM and SYSCTRL but includes DBADM authority on at least one of the databases that contains a table from which the view is created, the owner of the view can be any authorization ID if all of the following conditions are true:
  - The value of field DBADM CREATE AUTH was set to YES on panel DSNTIPP during DB2 installation.
  - The view is not based only on views.

  Note: The owner of the view must have the SELECT privilege on all tables and views in the CREATE VIEW statement, or, if the owner does not have the SELECT privilege on a table, the creator must have DBADM authority on the database that contains that table.
- If the privilege set lacks SYSADM, SYSCTRL, system DBADM, and DBADM authority, or if the authorization ID of the application plan or package fails to meet any of the previous conditions, the owner of the view must be the owner of the application plan or package.

If ROLE AS OBJECT OWNER is in effect, the schema qualifier must be the same as the role, unless the role has the CREATEIN privilege on the schema, SYSADM authority, system DBADM authority, or SYSCTRL authority.

If ROLE AS OBJECT OWNER is not in effect, one of the following rules applies:

- If the privilege set lacks the CREATIN privilege on the schema, SYSADM authority, system DBADM authority, or SYSCTRL authority, the schema qualifier (implicit or explicit) must be the same as one of the authorization ids of the process.
- If the privilege set includes system DBADM authority, SYSADM authority or SYSCTRL authority, the schema qualifier can be any valid schema name.

If the statement is dynamically prepared, the following rules apply:

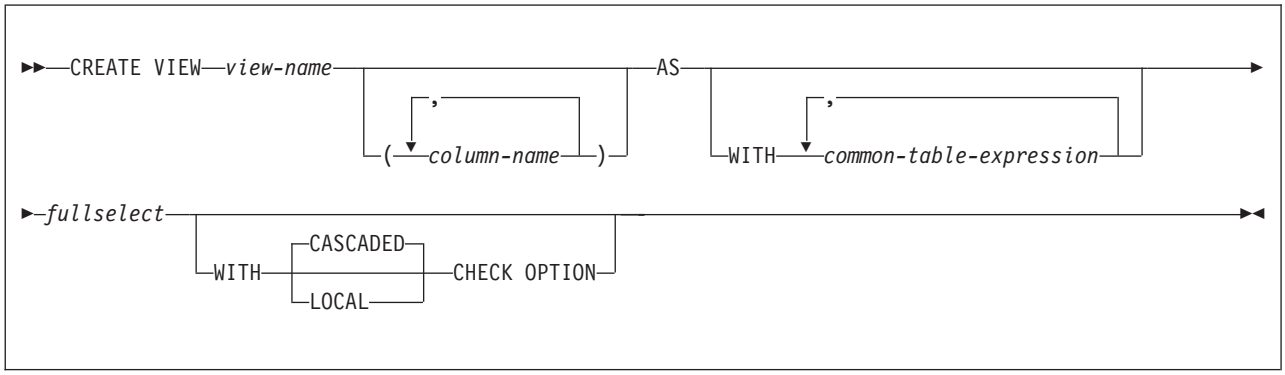
- If the SQL authorization ID of the process has SYSADM authority, the owner of the view can be any authorization ID. If that authorization ID has SYSCTRL but not SYSADM authority, the following is true: the owner of the view can be any authorization ID, provided the view does not refer to user tables or views in the first FROM clause of its defining fullselect. (It could refer instead, for example, to catalog tables or views thereof.)

  If the view satisfies the rules in the preceding paragraph, and if no errors are present in the CREATE statement, the view is created, even if the owner has no privileges at all on the tables and views identified in the fullselect of the view definition.
- If the SQL authorization ID of the process has system DBADM authority, the owner of the view can be any authorization ID. However, to create a view on a user table, either the owner of the view or the SQL authorization ID must have the SELECT privilege on all the tables or views in the CREATE VIEW statement.
- If SQL authorization ID of the process lacks system DBADM authority, SYSADM and SYSCTRL but includes DBADM authority on at least one of the databases that contains a table from which the view is created, the owner of the view can be different from the SQL authorization ID if all of the following conditions are true:
  - The value of field DBADM CREATE AUTH was set to YES on panel DSNTIPP during DB2 installation.
  - The view is not based only on views.

Note: The owner of the view must have the SELECT privilege on all tables and views in the CREATE VIEW statement, or, if the owner does not have the SELECT privilege on a table, the creator must have DBADM authority on the database that contains that table.

- If the SQL authorization ID of the process lacks SYSADM, SYSCTRL, system DBADM authority, or DBADM authority, or if the SQL authorization ID of the process fails to meet any of the previous conditions, only the authorization IDs of the process can own the view. In this case, the privilege set is the privileges that are held by the authorization ID selected for ownership.

Syntax



Description

view-name
Names the view. The name, including the implicit or explicit qualifier, must not identify a table, view, alias, or synonym that exists at the current server or a table that exists in the SYSIBM.SYSPENDINGOBJECTS catalog table. The unqualified name must not be the same as an existing synonym.

If the name is qualified, the name can be a two-part or three-part name. If a three-part name is used, the first part must match the value of the field DB2 LOCATION NAME of installation panel DSNTIPR at the current server. (If the current server is not the local DB2, this name is not necessarily the name in the CURRENT SERVER special register.)

column-name,...
Names the columns in the view. If you specify a list of column names, it must consist of as many names as there are columns in the result table of the fullselect. Each name must be unique and unqualified. If you do not specify a list of column names, the columns of the view inherit the names of the columns of the result table of the fullselect.

You must specify a list of column names if the result table of the fullselect has duplicate column names or an unnamed column (a column derived from a constant, function, or expression that was not given a name by the AS clause). For more details about unnamed columns, see the information about names of result columns under "select-clause" on page 765.

AS Identifies the view definition.

WITH *common-table-expression*
Defines a common table expression for use with the fullselect that follows. The fullselect must not contain a period specification. For an explanation of common table expression, see "common-table-expression" on page 820.

fullselect

Defines the view. At any time, the view consists of the rows that would result if the fullselect were executed.

The *fullselect* must conform to the following rules:

- The *fullselect* must not refer to any host variables or parameter markers (question marks).
- The *fullselect* must not refer to any declared temporary tables.
- The *fullselect* must not include an invocation of the UNPACK function.
- The *fullselect* must not contain a period specification.
- The FROM clause of the *fullselect* must not include a *data-change-table-reference*.
- The FROM clause of the *fullselect* must not include a view for which an INSTEAD OF trigger is defined.
- The outer SELECT list of the *fullselect* must not result in a column that is an array.

For an explanation of *fullselect*, see "fullselect" on page 811.

WITH CASCADED CHECK OPTION or WITH LOCAL CHECK OPTIONS

Specifies that every row that is inserted or updated through the view must conform to the definition of the view. A row that does not conform to the definition of the view is a row that cannot be retrieved using that view.

The CHECK OPTION clause must not be specified if the view is read-only, includes a subquery, references a function that is not deterministic or has an external action, or if the fullselect of the view refers to a created temporary table. If the CHECK OPTION clause is specified for an updatable view that does not allow inserts, it applies to updates only.

If the CHECK OPTION clause is omitted, the definition of the view is not used in the checking of any insert or update operations that use the view. Some checking might still occur during insert or update operations if the view is directly or indirectly dependent on another view that includes the CHECK OPTION clause. Because the definition of the view is not used, rows might be inserted or updated through the view that do not conform to the definition of the view.

The difference between the two forms of the check option, CASCADED and LOCAL, is meaningful only when a view is dependent on another view. The default is CASCADED. The view on which another view is directly or indirectly defined is an *underlying view*.

CASCADED

Update and insert operations on view V must satisfy the search conditions of view V and all underlying views, regardless of whether the underlying views were defined with a check option. Furthermore, every updatable view that is directly or indirectly defined on view V inherits those search conditions (the search conditions of view V and all underlying views of V) as a constraint on insert or update operations. WITH CASCADED CHECK OPTION must not be specified if a view on which the specified view definition is dependent has an INSTEAD OF trigger defined.

LOCAL

Update and insert operations on view V must satisfy the search conditions of view V and underlying views that are defined with a check option (either WITH CASCADED CHECK OPTION or WITH LOCAL CHECK OPTION). Furthermore, every updatable view that is directly or indirectly

defined on view V inherits those search conditions (the search conditions of view V and all underlying views of V that are defined with a check option) as a constraint on insert or update operations.

The LOCAL form of the CHECK option lets you update or insert rows that do not conform to the search condition of view V. You can perform these operations if the view is directly or indirectly defined on a view that was defined without a check option.

Table 134 illustrates the effect of using the default check option, CASCADED. The information in Table 134 is based on the following views:

- CREATE VIEW V1 AS SELECT COL1 FROM T1 WHERE COL1 > 10
- CREATE VIEW V2 AS SELECT COL1 FROM V1 WITH CASCADED CHECK OPTION
- CREATE VIEW V3 AS SELECT COL1 FROM V2 WHERE COL1 < 100

Table 134. Examples using default check option, CASCADED

SQL statement	Description of result
INSERT INTO V1 VALUES(5)	Succeeds because V1 does not have a check option and it is not dependent on any other view that has a check option.
INSERT INTO V2 VALUES(5)	Results in an error because the inserted row does not conform to the search condition of V1 which is implicitly is part of the definition of V2.
INSERT INTO V3 VALUES(5)	Results in an error because the inserted row does not conform to the search condition of V1.
INSERT INTO V3 VALUES(200)	Succeeds even though it does not conform to the definition of V3 (V3 does not have the view check option specified); it does conform to the definition of V2 (which does have the view check option specified).

The difference between CASCADED and LOCAL is shown best by example. Consider the following updatable views, where x and y represent either LOCAL or CASCADED:

V1 is defined on Table T0.
V2 is defined on V1 WITH x CHECK OPTION.
V3 is defined on V2.
V4 is defined on V3 WITH y CHECK OPTION.
V5 is defined on V4.

This example shows V1 as an *underlying view* for V2 and V2 as *dependent* on V1.

Table 135 shows the views in which search conditions are checked during an insert or update operation:

Table 135. Views in which search conditions are checked during insert and update operations

View used in INSERT or UPDATE operation	x = LOCAL y = LOCAL	x = CASCADED y = CASCADED	x = LOCAL y = CASCADED	x = CASCADED y = LOCAL
V1	None	None	None	None
V2	V2	V2, V1	V2	V2, V1
V3	V2	V2, V1	V2	V2, V1

Table 135. Views in which search conditions are checked during insert and update operations (continued)

View used in INSERT or UPDATE operation	x = LOCAL y = LOCAL	x = CASCADED y = CASCADED	x = LOCAL y = CASCADED	x = CASCADED y = LOCAL
V4	V4, V2	V4, V3, V2, V1	V4, V3, V2, V1	V4, V2, V1
V5	V4, V2	V4, V3, V2, V1	V4, V3, V2, V1	V4, V2, V1

Notes

Owner privileges

The owner of a view always acquires the SELECT privilege on the view and the authority to drop the view. If all of the privileges that are required to create the view are held with the GRANT option before the view is created, the owner of the view receives the SELECT privilege with the GRANT option. Otherwise, the owner receives the SELECT privilege without the GRANT option. For example, assume that a view definition also refers to a user-defined function. If the owner's EXECUTE privilege on the user-defined function is held without the GRANT option, the owner acquires the SELECT privilege on the view without the GRANT option.

The owner can also acquire INSERT, UPDATE, and DELETE privileges on the view. Acquiring these privileges is possible if the view is not "read-only", which means a single table of view is identified in the first FROM clause of the fullselect. For each privilege that the owner has on the identified table or view (INSERT, UPDATE, and DELETE) before the new view is created, the owner acquires that privilege on the view. The owner receives the privilege with the GRANT option if the privilege is held on the table or view with the GRANT option. Otherwise, the owner receives the privileges without the GRANT option.

With appropriate DB2 authority, a process can create views for those who have no authority to create the views themselves. The owner of such a view has the SELECT privilege on the view, without the GRANT option, and can drop the view.

For more information on the ownership of an object, see “Authorization, privileges, permissions, masks, and object ownership” on page 70.

Authorization for views created for other users:

When a process with appropriate authority creates a view for another user that does not have authorization for the underlying table or view, the SELECT privilege for the created view is implicitly granted to the user.

Considerations for row access control and column access control:

The view definition might reference a table for which row access control or column access control is activated. If the view definition references a table for which row access control or column access control is activated, the WITH CHECK OPTION clause must not be specified if the search conditions from the view or from the underlying views will be checked during an insert or update operation. Note that the WITH CHECK OPTION clause is ignored if such search conditions do not exist.

Read-only views:

A view is *read-only* if one or more of the following statements is true of its definition:

- The first FROM clause identifies more than one table or view, or identifies a table function, a nested table expression, a common table expression, or a collection-derived table.
- The first SELECT clause specifies the keyword DISTINCT.
- The outer fullselect contains a GROUP BY clause.
- The outer fullselect contains a HAVING clause.
- The first SELECT clause contains an aggregate function.
- It contains a subquery such that the base object of the outer fullselect, and of the subquery, is the same table.
- The first FROM clause identifies a read-only view.
- The first FROM clause identifies a system-maintained materialized query table.
- The outer fullselect is not a subselect (contains a set operator).

A read-only view cannot be the object of an SQL data change statement or a TRUNCATE statement. A view that includes GROUP BY or HAVING cannot be referred to in a subquery of a basic predicate.

Insertable views:

A view is insertable if an INSTEAD OF trigger for the insert operation has been defined for the view, or if at least one column of the view is updatable (independent of an INSTEAD OF trigger for update).

Considerations for implicitly hidden columns:

It is possible that the result table of the fullselect will include a column of a base table that is defined as implicitly hidden. This can occur when the implicitly hidden column is explicitly referenced in the fullselect of the view definition. However, the corresponding column of the view does not inherit the implicitly hidden attribute. Columns of a view cannot be defined as hidden.

Testing a view definition:

You can test the semantics of your view definition by executing SELECT * FROM *view-name*.

The two forms of a view definition:

Both the source and the operational form of a view definition are stored in the DB2 catalog. Those two forms are not necessarily equivalent because the operational form reflects the state that exists when the view is created. For example, consider the following statement:

```
CREATE VIEW V AS SELECT * FROM S;
```

In this example, S is a synonym or alias for A.T, which is a table with columns C1, C2, and C3. The operational form of the view definition is equivalent to:

```
SELECT C1, C2, C3 FROM A.T;
```

Adding columns to A.T using ALTER TABLE and dropping S does not affect the operational form of the view definition. Thus, if columns are added to A.T or if S is redefined, the source form of the view definition can be misleading.

View restrictions:

A view definition cannot contain references to remote objects. A view definition cannot map to more than 15 base table instances. A view definition cannot reference a declared global temporary table.

Restrictions involving pending definition changes:
CREATE VIEW is not allowed if the view references a column on which there are pending definition changes.

Considerations for inline LOB columns:
If the view references a table that contains an inline LOB column and DB2 determines that the inline attribute can be passed on to the view, the view will then inherit the inline attribute, otherwise the inline attribute is not inherited by the view.

Considerations for XML columns:
If the view has an XML column and the column of the underlying base table for the view has an XML type modifier, the view column has the same type modifier. However, if there is an instead of trigger defined on the view, validation of the column, according to XML schemas in the type modifier, is not enforced during insert or update to this view.

Examples

Example 1: Create the view DSN8B10.VPROJRE1. PROJNO, PROJNAME, PROJDEP, RESPEMP, FIRSTNME, MIDINIT, and LASTNAME are column names. The view is a join of tables and is therefore read-only.

```
CREATE VIEW DSN8B10.VPROJRE1
  (PROJNO,PROJNAME,PROJDEP,RESPEMP,
   FIRSTNME,MIDINIT,LASTNAME)
  AS SELECT ALL
  PROJNO,PROJNAME,DEPTNO,EMPNO,
  FIRSTNME,MIDINIT,LASTNAME
  FROM DSN8B10.PROJ, DSN8B10.EMP
  WHERE RESPEMP = EMPNO;
```

In the example, the WHERE clause refers to the column EMPNO, which is contained in one of the base tables but is not part of the view. In general, a column named in the WHERE, GROUP BY, or HAVING clause need not be part of the view.

Example 2: Create the view DSN8B10.FIRSTQTR that is the UNION ALL of three fullselects, one for each month of the first quarter of 2000. The common names are SNO, CHARGES, and DATE.

```
 CREATE VIEW DSN8B10.FIRSTQTR (SNO, CHARGES, DATE) AS
SELECT SNO, CHARGES, DATE
FROM MONTH1
WHERE DATE BETWEEN '01/01/2000' and '01/31/2000'
  UNION All
SELECT SNO, CHARGES, DATE
FROM MONTH2
WHERE DATE BETWEEN '02/01/2000' and '02/29/2000'
  UNION All
SELECT SNO, CHARGES, DATE
FROM MONTH3
WHERE DATE BETWEEN '03/01/2000' and '03/31/2000';
```

DECLARE CURSOR

The DECLARE CURSOR statement defines a cursor.

Invocation

This statement can only be embedded in an application program. It is not an executable statement. It must not be specified in Java.

Authorization

For each table or view identified in the SELECT statement of the cursor, the privilege set must include at least one of the following:

- The SELECT privilege
- Ownership of the object
- DBADM authority for the corresponding database (tables only)
- SYSADM authority
- SYSCTRL authority (catalog tables only)
- DATAACCESS authority

If the *select-statement* contains an SQL data change statement, the authorization requirements of that statement also apply to the DECLARE CURSOR statement.

The SELECT statement of the cursor is one of the following:

- The prepared select statement identified by *statement-name*
- The specified *select-statement*

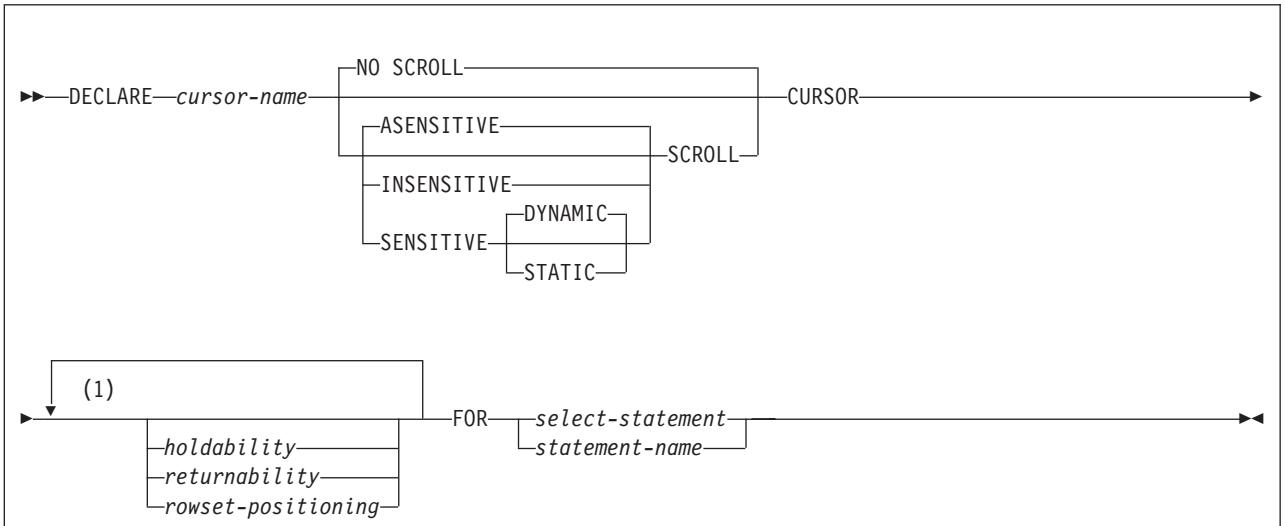
If statement-name is specified:

- The privilege set is determined by the DYNAMICRULES behavior in effect (run, bind, define, or invoke) and is summarized in Table 94 on page 841. (For more information on these behaviors, including a list of the DYNAMICRULES bind option values that determine them, see "Authorization IDs and dynamic SQL" on page 75.)
- The authorization check is performed when the SELECT statement is prepared.
- The cursor cannot be opened unless the SELECT statement is successfully prepared.

If select-statement is specified:

- The privilege set consists of the privileges that are held by the authorization ID of the owner of the plan or package.
- If the plan or package is bound with VALIDATE(BIND), the authorization check is performed at bind time, and the bind is unsuccessful if any required privilege does not exist.
- If the plan or package is bound with VALIDATE(RUN), an authorization check is performed at bind time, but all required privileges need not exist at that time. If all privileges exist at bind time, no authorization checking is performed when the cursor is opened. If any privilege does not exist at bind time, an authorization check is performed the first time the cursor is opened within a unit of work. The OPEN is unsuccessful if any required privilege does not exist.

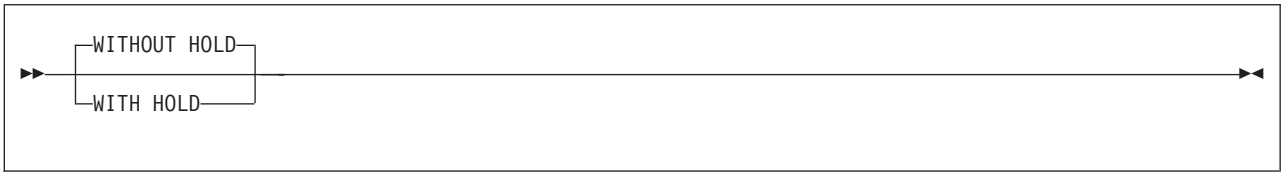
Syntax



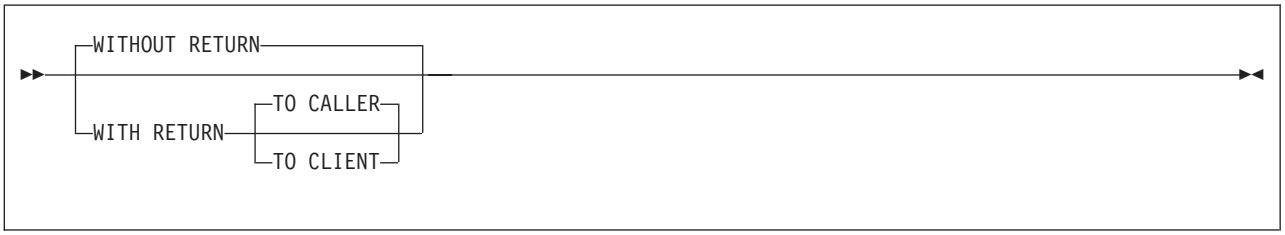
Notes:

1 The same clause must not be specified more than once.

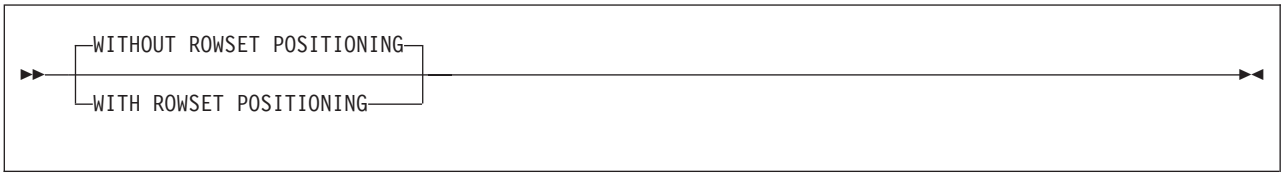
holdability:



returnability:



rowset-positioning:



Description

cursor-name
Names the cursor. The name must not identify a cursor that has already been

declared in the source program. The name is usually VARCHAR(128); however, if the cursor is defined WITH RETURN, the name is limited to VARCHAR(30).

NO SCROLL or SCROLL
Specifies whether the cursor is scrollable or not scrollable.

NO SCROLL
Specifies that the cursor is not scrollable. This is the default.

SCROLL
Specifies that the cursor is scrollable. For a scrollable cursor, whether the cursor has sensitivity to inserts, updates, or deletes depends on the cursor sensitivity option in effect for the cursor. If a sensitivity option is not specified, ASENSITIVE is the default.

ASENSITIVE
Specifies that the cursor should be as sensitive as possible. This is the default.

A cursor that defined as ASENSITIVE will be either insensitive or sensitive dynamic; it will not be sensitive static. For information about how the effective sensitivity of the cursor is returned to the application with the GET DIAGNOSTICS statement or in the SQLCA, see "OPEN" on page 1775.

INSENSITIVE
Specifies that the cursor does not have sensitivity to inserts, updates, or deletes that are made to the rows underlying the result table. As a result, the size of the result table, the order of the rows, and the values for each row do not change after the cursor is opened. In addition, the cursor is read-only. The SELECT statement or *attribute-string* of the PREPARE statement cannot contain a FOR UPDATE clause, and the cursor cannot be used for positioned updates or deletes.

SENSITIVE
Specifies that the cursor has sensitivity to changes that are made to the database after the result table is materialized. The cursor is always sensitive to updates and deletes that are made using the cursor (that is, positioned updates and deletes using the same cursor). When the current value of a row no longer satisfies the *select-statement* or *statement-name*, that row is no longer visible through the cursor. When a row of the result table is deleted from the underlying base table, the row is no longer visible through the cursor.

If DB2 cannot make changes visible to the cursor, then an error is issued at bind time for OPEN CURSOR. DB2 cannot make changes visible to the cursor when the cursor implicitly becomes read-only. For example, when the result table must be materialized, as when the FROM clause of the SELECT statement contains more than one table or view. The current list of conditions that result in an implicit read-only cursor can be found in Read-only cursors.

The default is DYNAMIC.

DYNAMIC
Specifies that the result table of the cursor is dynamic, meaning that the size of the result table might change after the cursor is opened as rows are inserted into or deleted from the underlying table, and the order of the rows might change. Rows that are inserted, deleted, or updated by statements that are executed by the same application process as the cursor are visible to the cursor

immediately. Rows that are inserted, deleted, or updated by statements that are executed by other application processes are visible only after the statements are committed. If a column for an ORDER BY clause is updated via a cursor or any means outside the process, the next FETCH statement behaves as if the updated row was deleted and re-inserted into the result table at its correct location. At the time of a positioned update, the cursor is positioned before the next row of the original location and there is no current row, making the row appear to have moved.

If a SENSITIVE DYNAMIC cursor is not possible, an error is returned. For example, if a temporary table is needed an error is returned. The SELECT statement of a cursor that is defined as SENSITIVE DYNAMIC cannot contain an SQL data change statement.

The FETCH FIRST *n* ROWS ONLY clause must not be specified for the outermost fullselect for a sensitive dynamic cursor.

STATIC
Specifies that the size of the result table and the order of the rows do not change after the cursor is opened. Rows inserted into the underlying table are not added to the result table regardless of how the rows are inserted. Rows in the result table do not move if columns in the ORDER BY clause are updated in rows that have already been materialized. Positioned updates and deletes are allowed if the result table is updatable. The SELECT statement of a cursor that is defined as SENSITIVE STATIC cannot contain an SQL data change statement.

A STATIC cursor has visibility to changes made by *this* cursor using positioned updates or deletes. Committed changes made outside this cursor are visible with the SENSITIVE option of the FETCH statement. A FETCH SENSITIVE can result in a *hole* in the result table (that is, a difference between the result table and its underlying base table). If an updated row in the base table of a cursor no longer satisfies the predicate of its SELECT statement, an update hole occurs in the result table. If a row of a cursor was deleted in the base table, a delete hole occurs in the result table. When a FETCH SENSITIVE detects an update hole, no data is returned (a warning is issued), and the cursor is left positioned on the update hole. When a FETCH SENSITIVE detects a delete hole, no data is returned (a warning is issued), and the cursor is left positioned on the delete hole.

Updates through a cursor result in an automatic re-fetch of the row. This re-fetch means that updates can create a hole themselves. The re-fetched row also reflects changes as a result of triggers updating the same row. It is important to reflect these changes to maintain the consistency of data in the row.

Using a function that is not deterministic (built-in or user-defined) in the WHERE clause of the *select-statement* or *statement-name* of a SENSITIVE STATIC cursor can cause misleading results. This situation occurs because DB2 constructs a temporary result table and retrieves rows from this table for FETCH INSENSITIVE statements. When DB2 processes a FETCH SENSITIVE statement, rows are fetched from the underlying table and predicates are re-evaluated. Using a function that is not deterministic can yield a

different result on each FETCH SENSITIVE of the same row, which could also result in the row no longer being considered a match.

A FETCH INSENSITIVE on a SENSITIVE STATIC SCROLL cursor is not sensitive to changes made outside the cursor, unless a previous FETCH SENSITIVE has already refreshed that row; however, positioned updates and delete changes with the cursor are visible.

STATIC cursors are insensitive to insertions.

WITHOUT HOLD or WITH HOLD
Specifies whether the cursor should be prevented from being closed as a consequence of a commit operation.

WITHOUT HOLD
Does not prevent the cursor from being closed as a consequence of a commit operation. This is the default.

WITH HOLD
Prevents the cursor from being closed as a consequence of a commit operation. A cursor declared with WITH HOLD is closed at commit time if one of the following is true:

- The connection associated with the cursor is in the release pending status.
- The bind option DISCONNECT(AUTOMATIC) is in effect.
- The environment is one in which the option WITH HOLD is ignored.

When WITH HOLD is specified, a commit operation commits all of the changes in the current unit of work. For example, with a non-scrollable cursor, an initial FETCH statement is needed after a COMMIT statement to position the cursor on the row that follows the row that the cursor was positioned on before the commit operation.

WITH HOLD has no effect on an SQL data change statement within a SELECT statement. When a COMMIT is issued, the changes caused by the SQL data change statement are committed, regardless of whether or not the cursor is declared WITH HOLD.

All cursors are implicitly closed by a connect (Type 1) or rollback operation. A cursor is also implicitly closed by a commit operation if WITH HOLD is ignored or not specified.

Cursors that are declared with WITH HOLD in CICS or in IMS non-message-driven programs will not be closed by a rollback operation if the cursor was opened in a previous unit of work and no changes have been made to the database in the current unit of work. The cursor cannot be closed because CICS and IMS do not broadcast the rollback request to DB2 for a null unit of work.

If a cursor is closed before the commit operation, the effect is the same as if the cursor was declared without the option WITH HOLD.

WITH HOLD is ignored in IMS message driven programs (MPP, IFP, and message-driven BMP). WITH HOLD maintains the cursor position in a CICS pseudo-conversational program until the end-of-task (EOT).

For details on restrictions that apply to declaring cursors with WITH HOLD, see *DB2 Application Programming and SQL Guide*.

WITHOUT RETURN or WITH RETURN
Specifies whether the result table of the cursor is intended to be used as a

result set that will be returned from a procedure. If *statement-name* is specified, the default is the corresponding prepare attribute of the statement. Otherwise, the default is **WITHOUT RETURN**.

WITHOUT RETURN
Specifies that the result table of the cursor is not intended to be used as a result set that will be returned from a procedure.

WITH RETURN
Specifies that the result table of the cursor is intended to be used as a result set that will be returned from a procedure. WITH RETURN is relevant only if the DECLARE CURSOR statement is contained within the source code for a procedure. In other cases, the precompiler might accept the clause, but it has not effect.

When a cursor that is declared using the WITH RETURN TO CALLER clause remains open at the end of a program or routine, that cursor defines a result set from the program or routine. Use the CLOSE statement to close a cursor that is not intended to be a result set from the program or routine. Although DB2 will automatically close any cursors that are not declared using with a WITH RETURN clause, the use of the CLOSE statement is recommended to increase the portability of applications.

For non-scrollable cursors, the result set consists of all rows from the current cursor position to the end of the result table. For scrollable cursors, the result set consists of all rows of the result table.

TO CALLER
Specifies that the cursor can return a result set to the caller of the procedure. The caller is the program or routine that executed the SQL CALL statement that invokes the procedure that contains the DECLARE CURSOR statement. For example, if the caller is a procedure, the result set, is returned to the procedure. If the caller is a client application, the result set is returned to the client application.

If the statement is contained within the source code for a procedure, WITH RETURN TO CALLER specifies that the cursor can be used as a result set cursor. A result set cursor is used when the result table of a cursor is to be returned from a procedure. Specifying TO CALLER is optional.

In other cases, the clause is ignored and the cursor cannot be used as a result set cursor.

TO CLIENT
Specifies that the cursor can return a result set to the client application. This cursor is invisible to any intermediate nested procedures. If a function or trigger calls the procedure (either directly or indirectly), the result set cannot be returned to the client and the cursor will be closed after the procedure finishes.

rowset-positioning
Specifies whether multiple rows of data can be accessed as a rowset on a single FETCH statement for the cursor. The default is WITHOUT ROWSET POSITIONING.

WITHOUT ROWSET POSITIONING
Specifies that the cursor can be used only with row-positioned FETCH statements. The cursor is to return a single row for each FETCH statement and the FOR *n* ROWS clause cannot be specified on a FETCH statement for this cursor. WITHOUT ROWSET POSITIONING or single row access

refers to how data is fetched from the database engine. For remote access, data might be blocked and returned to the client in blocks.

WITH ROWSET POSITIONING
Specifies that the cursor can be used with either row-positioned or rowset-positioned FETCH statements. This cursor can be used to return either a single row or multiple rows, as a rowset, with a single FETCH statement. ROWSET POSITIONING refers to how data is fetched from the database engine. For remote access, if any row qualifies, at least 1 row is returned as a rowset. The size of the rowset depends on the number of rows specified on the FETCH statement and on the number of rows that qualify. Data might be blocked and returned to the client in blocks.

select-statement
Specifies the result table of the cursor. See "select-statement" on page 819 for an explanation of *select-statement*.

The *select-statement* must not include parameter markers (except for REXX), but can include references to host variables. In host languages, other than REXX, the declarations of the host variables must precede the DECLARE CURSOR statement in the source program. In REXX, parameter markers must be used in place of host variables and the statement must be prepared.

The USING clause of the OPEN statement can be used to specify host variables that will override the values of the host variables or parameter markers that are specified as part of the statement in the DECLARE CURSOR statement.

The *select-statement* of the cursor must not contain an SQL data change statement if the cursor is defined as SENSITIVE DYNAMIC or SENSITIVE STATIC.

The outer select list of the *select-statement* of a scrollable cursor must not be an array value.

statement-name
Identifies the prepared *select-statement* that specifies the result table of the cursor whenever the cursor is opened. The *statement-name* must not be identical to a statement name specified in another DECLARE CURSOR statement of the source program. For an explanation of prepared SELECT statements, see "PREPARE" on page 1781.

The prepared *select-statement* of the cursor must not contain an SQL data change statement if the cursor is defined as SENSITIVE DYNAMIC or SENSITIVE STATIC.

Notes

A cursor in the open state designates a result table and a position relative to the rows of that table. The table is the result table specified by the SELECT statement of the cursor.

Read-only cursors: If the result table is *read-only*, the cursor is *read-only*.The cursor that references a view with instead of triggers are read-only since positioned UPDATE and positioned DELETE statements are not allowed using those cursors. The result table is *read-only* if one or more of the following statements is true about the SELECT statement of the cursor:

- The first FROM clause identifies or contains any of the following:
  - More than one table or view
  - A catalog table with no updatable columns

  - A read-only view
  - A nested table expression
  - A table function
  - A system-maintained materialized query table
  - A single table that is a system-period temporal table, and a period specification for SYSTEM_TIME is used
  - A single view that directly or indirectly references a system-period temporal table in the FROM clause of the outer fullselect of the view definition, and a period specification for SYSTEM_TIME is used
- The first SELECT clause specifies the keyword DISTINCT, contains an aggregate function, or uses both
- It contains an SQL data change statement
- The outer subselect contains a GROUP BY clause, a HAVING clause, or both clauses
- It contains a subquery such that the base object of the outer subselect, and of the subquery, is the same table
- Any of the following operators or clauses are specified:
  - A set operator
  - An ORDER BY clause (except when the cursor is declared as SENSITIVE STATIC scrollable)
  - A FOR READ ONLY clause
- It is executed with isolation level UR and a FOR UPDATE clause is not specified.

If the result table is not *read-only*, the cursor can be used to update or delete the underlying rows of the result table.

Tables for which row or column access controls are enforced: The *select-statement* of the cursor can reference a table for which row or column access controls are enforced. The row or column access controls do not effect the determination of whether the cursor is read-only and do not effect the cursor sensitivity.

Work file database requirement for static scrollable cursors: To use a static scrollable cursor, you must first create a work file database and at least one table space with a 32KB page size in this database because a static scrollable cursor requires a temporary table for its result table while the cursor is open. DB2 chooses a table space to use for the temporary result table. Dynamic scrollable cursors do not require a declared temporary table.

For static scrollable cursor declarations that contain empty strings, DB2 assigns one byte in the temporary table space for each empty string. The following example shows a scrollable cursor declaration with an empty string:

```
EXEC SQL DECLARE CSROWSTAT SENSITIVE STATIC SCROLL CURSOR
  WITH ROWSET POSITIONING WITH HOLD FOR
  SELECT ID1,''
  FROM TB;
```

Cursors in COBOL and Fortran programs: In COBOL and Fortran source programs, the DECLARE CURSOR statement must precede all statements that explicitly refer to the cursor by name. This rule does not necessarily apply to the other host languages because the precompiler provides a two-pass option for these languages. This rule applies to other host languages if the two-pass option is not used.

Cursors in REXX: If host variables are used in a DECLARE CURSOR statement within a REXX procedure, the DECLARE CURSOR statement must be the object of a PREPARE and EXECUTE.

Scope of a cursor: The scope of *cursor-name* is the source program in which it is defined; that is, the application program submitted to the precompiler. Thus, you can only refer to a cursor by statements that are precompiled with the cursor declaration. For example, a COBOL program called from another program cannot use a cursor that was opened by the calling program. Furthermore, a cursor defined in a Fortran subprogram can only be referred to in that subprogram. Cursors that specify WITH RETURN in a procedure and are left open are returned as result sets.

Although the scope of a cursor is the program in which it is declared, each package (or DBRM of a plan) created from the program includes a separate instance of the cursor, and more than one instance of the cursor can be used in the same execution of the program. For example, assume a program is precompiled with the CONNECT(2) option and its DBRM is used to create a package at location X and a package at location Y. The program contains the following SQL statements:

```
DECLARE C CURSOR FOR ...
CONNECT TO X
OPEN C
FETCH C INTO ...
CONNECT TO Y
OPEN C
FETCH C INTO ...
```

The second OPEN C statement does not cause an error because it refers to a different instance of cursor C. The same notion applies to a single location if the packages are in different collections.

A SELECT statement is evaluated at the time the cursor is opened. If the same cursor is opened, closed, and then opened again, the results can be different. If the SELECT statement of the cursor contains CURRENT DATE, CURRENT TIME or CURRENT TIMESTAMP, all references to these special registers yields the same respective datetime value on each FETCH operation. The value is determined when the cursor is opened. Multiple cursors using the same SELECT statement can be opened concurrently. They are each considered independent activities.

Blocking of data: To process data more efficiently, DB2 might block data for read-only cursors. If a cursor is not going to be used in a positioned UPDATE or positioned DELETE statement, define the cursor as FOR READ ONLY.

Positioned deletes and isolation level UR: Specify FOR UPDATE if you want to use the cursor for a positioned DELETE and the isolation level is UR because of a BIND option. In this case, the isolation level is CS.

Returning a result set from a stored procedure: A cursor that is declared in a stored procedure returns a result set when all of the following conditions are true:

- The cursor is declared with the WITH RETURN option. In a distributed environment, blocks of each result set of the cursor's data are returned with the CALL statement reply.
- The cursor is left open after exiting from the stored procedure. A cursor declared with the SCROLL option must be left positioned *before* the first row before exiting from the stored procedure.

- The cursor is declared with the WITH HOLD option if the stored procedure is defined to commit on return.

The result set is the set of all rows after the current position of the cursor after exiting the stored procedure. The result set is assumed to be read-only. If that same procedure is reinvoked, open result set cursors for a stored procedure at a given site are automatically closed by the database management system.

Scrollable cursors specified with user-defined functions: A row can be fetched more than once with a scrollable cursor. Therefore, if a scrollable cursor is defined with a function that is not deterministic in the select list of the cursor, a row can be fetched multiple times with different results for each fetch. (However, the value of a funtion that is not deterministic in the WHERE clause of a scrollable cursor is captured when the cursor is opened and remains unchanged until the cursor is closed.) Similarly, if a scrollable cursor is defined with a user-defined function with external action, the action is executed with every fetch.

Multiple instances of a cursor that is defined with RETURN TO CLIENT: If the cursor is declared in a native SQL procedure, a cursor that is declared as WITH RETURN TO CLIENT can be opened even when a cursor with the same name is already in the open state. In this case, the already open cursor becomes a result set cursor and is no longer accessible by using its cursor name. A new cursor is opened and becomes accessible by using the cursor name. When a CLOSE statement is issued, the last instance of the cursor will be closed. Closing the new cursor does not make the cursor that was previously accessible by that name accessible by the cursor name again. Cursors that become result set cursors in this way cannot be accessed at the server and can be processed only at the client.

Examples

The statements in the following examples are assumed to be in PL/I programs.

Example 1: Declare C1 as the cursor of a query to retrieve data from the table DSN8B10.DEPT. The query itself appears in the DECLARE CURSOR statement.

```
EXEC SQL DECLARE C1 CURSOR FOR
   SELECT DEPTNO, DEPTNAME, MGRNO
   FROM DSN8B10.DEPT
   WHERE ADMRDEPT = 'A00';
```

Example 2: Declare C1 as the cursor of a query to retrieve data from the table DSN8810.DEPT. Assume that the data will be updated later with a searched update and should be locked when the query executes. The query itself appears in the DECLARE CURSOR statement.

```
EXEC SQL DECLARE C1 CURSOR FOR
   SELECT DEPTNO, DEPTNAME, MGRNO
   FROM DSN8B10.DEPT
   WHERE ADMRDEPT = 'A00'
   FOR READ ONLY WITH RS USE AND KEEP EXCLUSIVE LOCKS;
```

Example 3: Declare C2 as the cursor for a statement named STMT2.

```
EXEC SQL DECLARE C2 CURSOR FOR STMT2;
```

Example 4: Declare C3 as the cursor for a query to be used in positioned updates of the table DSN8B10.EMP. Allow the completed updates to be committed from time to time without closing the cursor.

```
EXEC SQL DECLARE C3 CURSOR WITH HOLD FOR
  SELECT * FROM DSN8B10.EMP
    FOR UPDATE OF WORKDEPT, PHONENO, JOB, EDLEVEL, SALARY;
```

Instead of specifying which columns should be updated, you could use a FOR UPDATE clause without the names of the columns to indicate that all updatable columns are updated.

Example 5: In stored procedure SP1, declare C4 as the cursor for a query of the table DSN8B10.PROJ. Enable the cursor to return a result set to the caller of SP1, which performs a commit on return.

```
EXEC SQL DECLARE C4 CURSOR WITH HOLD WITH RETURN FOR
   SELECT PROJNO, PROJNAME
   FROM DSN8B10.PROJ
   WHERE DEPTNO = 'A01';
```

Example 6: In the following example, the DECLARE CURSOR statement associates the cursor name C5 with the results of the SELECT and specifies that the cursor is scrollable. C5 allows positioned updates and deletes because the result table can be updated.

```
EXEC SQL DECLARE C5 SENSITIVE STATIC SCROLL CURSOR FOR
   SELECT DEPTNO, DEPTNAME, MGRNO
   FROM DSN8B10.DEPT
   WHERE ADMRDEPT = 'A00';
```

Example 7: In the following example, the DECLARE CURSOR statement associates the cursor name C6 with the results of the SELECT and specifies that the cursor is scrollable.

```
EXEC SQL DECLARE C6 INSENSITIVE SCROLL CURSOR FOR
   SELECT DEPTNO, DEPTNAME, MGRNO
   FROM DSN8B10.DEPT
   WHERE DEPTNO;
```

Example 8: The following example illustrates how an application program might use dynamic scrollable cursors. First create and populate a table.

```
CREATE TABLE ORDER
      (ORDERNUM  INTEGER,
       CUSTNUM   INTEGER,
       CUSTNAME  VARCHAR(20),
       ORDERDATE CHAR(8),
       ORDERAMT  DECIMAL(8,3),
       COMMENTS  VARCHAR(20));
```

Populate the table by inserting or loading about 500 rows.

```
EXEC SQL DECLARE CURSOR ORDERSCROLL
   SENSITIVE DYNAMIC SCROLL FOR
   SELECT ORDERNUM, CUSTNAME, ORDERAMT, ORDERDATE FROM ORDER
   WHERE ORDERAMT > 1000
   FOR UPDATE OF COMMENTS;
```

Open the scrollable cursor.

```
OPEN CURSOR ORDERSCROLL;
```

Fetch forward from the scrollable cursor.

```
-- Loop-to-fill-screen
      -- do 10 times
       FETCH FROM ORDERSCROLL INTO :HV1, :HV2, :HV3, :HV4;
      -- end
```

Fetch RELATIVE from the scrollable cursor.

```
-- Skip-forward-100-rows
      FETCH RELATIVE +100
       FROM ORDERSCROLL INTO :HV1, :HV2, :HV3, :HV4;
-- Skip-backward-50-rows
      FETCH RELATIVE -50
       FROM ORDERSCROLL INTO :HV1, :HV2, :HV3, :HV4;
```

Fetch ABSOLUTE from the scrollable cursor.

```
-- Re-read-the-third-row
      FETCH ABSOLUTE +3
       FROM ORDERSCROLL INTO :HV1, :HV2, :HV3, :HV4;
```

Fetch RELATIVE from scrollable cursor.

```
-- Read-the-third-row-from current position
      FETCH SENSITIVE RELATIVE +3
       FROM ORDERSCROLL INTO :HV1, :HV2, :HV3, :HV4;
```

Do a positioned update through the scrollable cursor.

```
-- Update-the-current-row
      UPDATE ORDER SET COMMENTS = "Expedite"
       WHERE CURRENT OF ORDERSCROLL;
```

Close the scrollable cursor.

```
CLOSE CURSOR ORDERSCROLL;
```

Example 9: Declare C1 as the cursor of a query to retrieve a rowset from the table DEPT. The prepared statement is MYCURSOR.

```
EXEC SQL DECLARE C1 CURSOR
   WITH ROWSET POSITIONING FOR MYCURSOR;
```

DECLARE GLOBAL TEMPORARY TABLE

The DECLARE GLOBAL TEMPORARY TABLE statement defines a declared temporary table for the current application process. The declared temporary table resides in the work file database and its description does not appear in the system catalog. It is not persistent and cannot be shared with other application processes. Each application process that defines a declared temporary table of the same name has its own unique description and instance of the temporary table. When the application process terminates, the temporary table is dropped.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

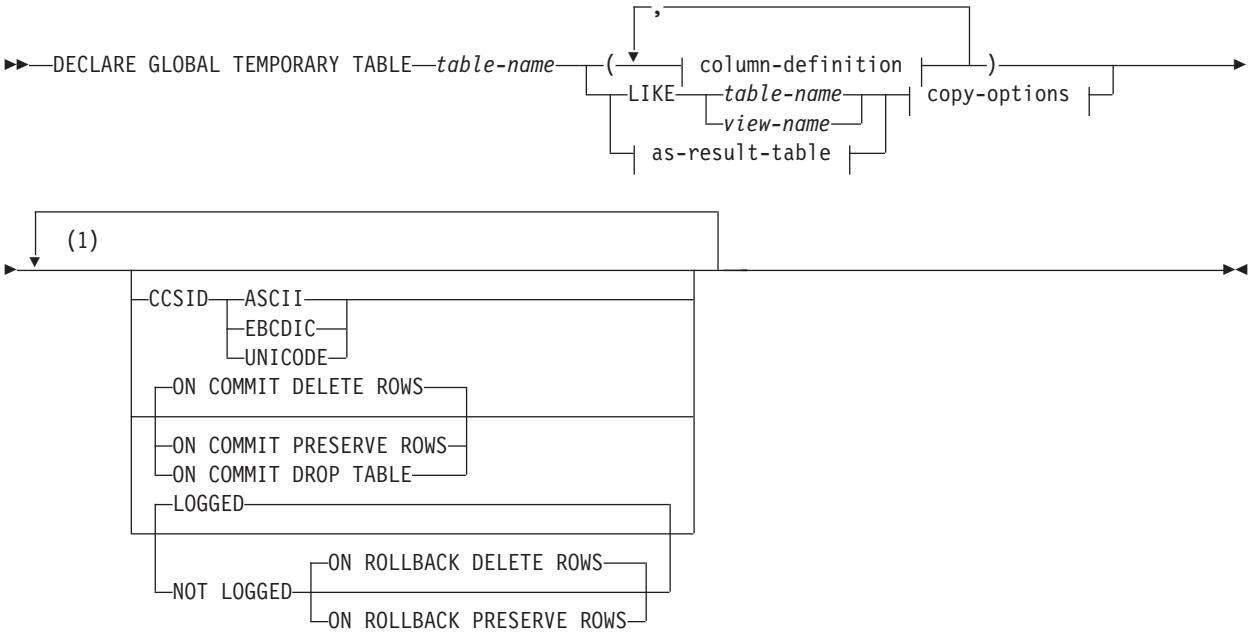
None are required, unless the LIKE clause is specified when additional privileges might be required.

PUBLIC implicitly has the following privileges without GRANT authority for declared temporary tables:

- The CREATETAB privilege to define a declared temporary table in the database that is defined AS WORKFILE, which is the database for declared temporary tables.
- The USE privilege to use the table spaces in the database that is defined as WORKFILE.
- All table privileges on the table and authority to drop the table. (Table privileges for a declared temporary table cannot be granted or revoked.)

These implicit privileges are not recorded in the DB2 catalog and cannot be revoked.

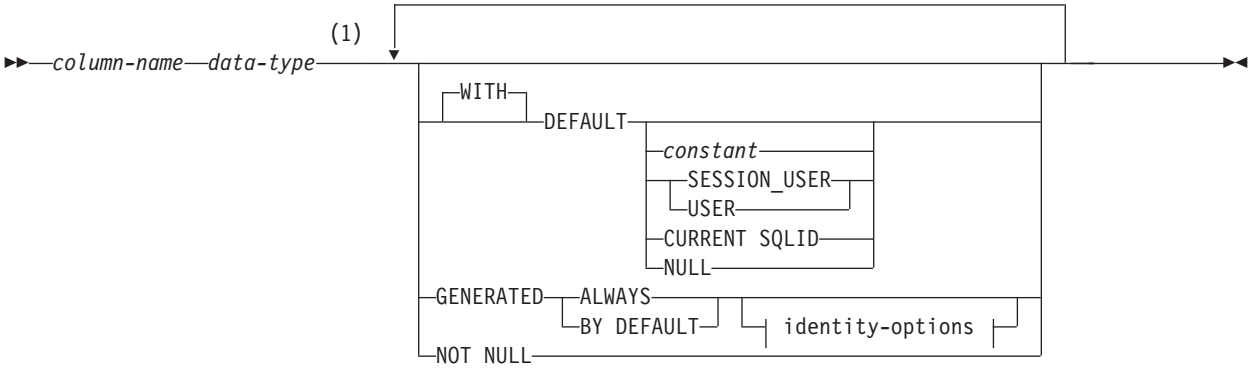
Syntax



Notes:

1 The same clause must not be specified more than one time.

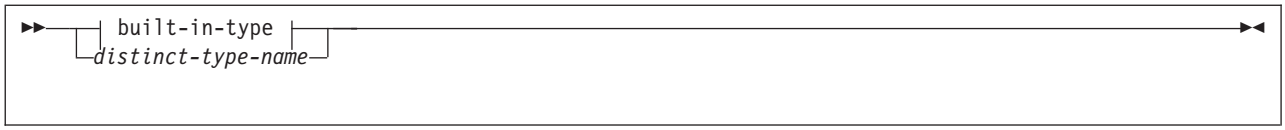
column-definition:



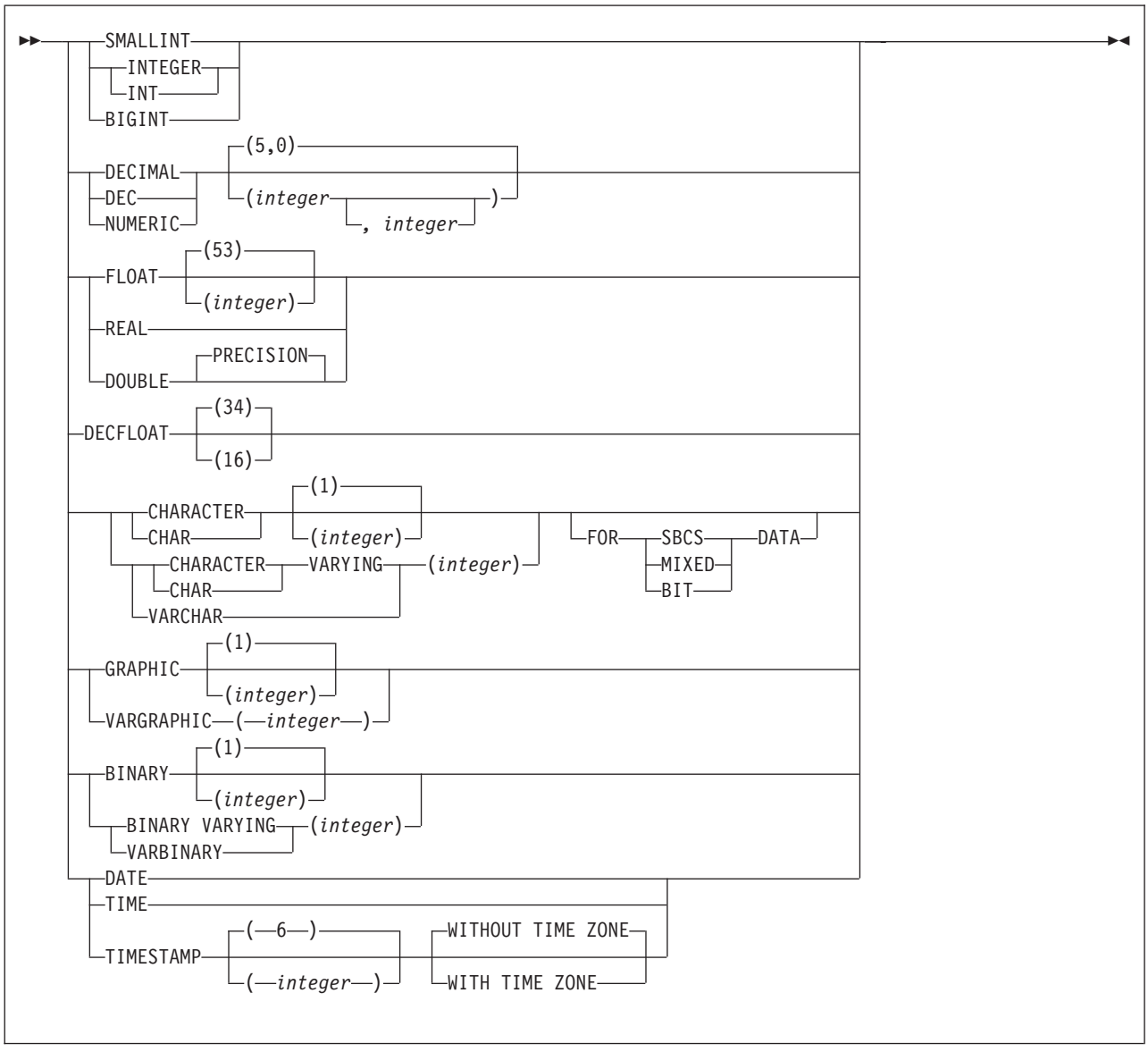
Notes:

1 The same clause must not be specified more than once.

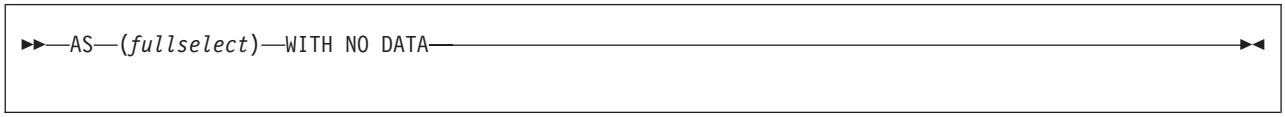
data-type:



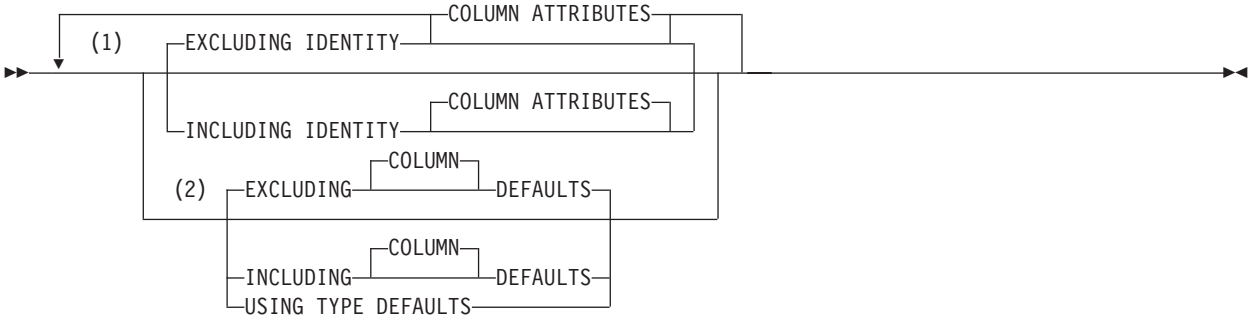
built-in-type:



as-result-table:



copy-options:

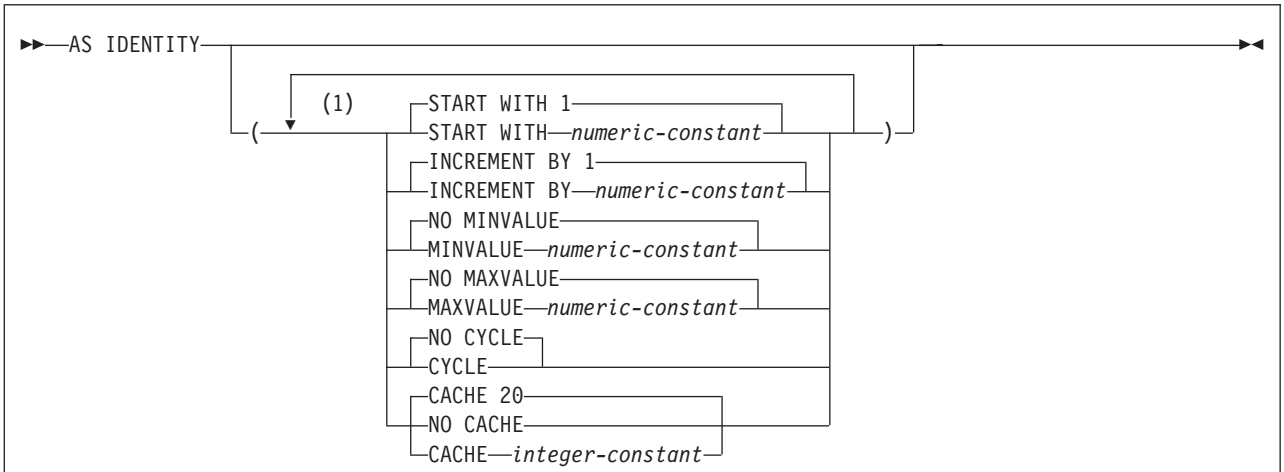


Notes:

1 These clauses can be specified in any order and must not be specified more than one time.

2 EXCLUDING COLUMN DEFAULTS, INCLUDING COLUMN DEFAULTS, and USING TYPE DEFAULTS must not be specified with the LIKE clause.

identity-options:



Notes:

1 Separator commas can be specified between the attributes when an identity column is defined

Description

table-name

Names the temporary table. The qualifier, if specified explicitly, must be SESSION. If the qualifier is not specified, it is implicitly defined to be SESSION.

If a table, view, synonym, or alias already exists with the same name and an implicit or explicit qualifier of SESSION:

- The declared temporary table is still defined with SESSION.*table-name*. An error is not issued because the resolution of a declared temporary table name does not include the persistent and shared names in the DB2 catalog tables.

- Any references to SESSION.*table-name* will resolve to the declared temporary table rather than to any existing SESSION.*table-name* whose definition is persistent and is in the DB2 catalog tables.

column-definition
Defines the attributes of a column for each instance of the table. The number of columns defined must not exceed 750. The maximum record size must not exceed 32683 bytes. The maximum row size must not exceed 32675 bytes (8 bytes less than the maximum record size).

column-name
Names the column. The name must not be qualified and must not be the same as the name of another column in the table.

data-type
Specifies the data type of the column. The data type can be any built-in data type that can be specified for the CREATE TABLE statement except for a LOB (BLOB, CLOB, and DBCLOB), ROWID, or XML type. The FOR *subtype* DATA clause can be specified as part of *data-type*. For more information on the data types and the rules that apply to them, see built-in-type.

DEFAULT
Specifies a default value for the column. This clause must not be specified more than once in the same *column-definition*.

Omission of NOT NULL and DEFAULT from a *column-definition* is an implicit specification of DEFAULT NULL.

If DEFAULT is specified without a value after it, the default value of the column depends on the data type of the column, as follows:

Data type
Default value
Numeric
0
Fixed-length character string
A string of blanks
Fixed-length graphic string
A string of blanks
Fixed-length binary string
Hexadecimal zeros
Varying-length string
A string of length 0
Date CURRENT DATE
Time CURRENT TIME
Timestamp
CURRENT TIMESTAMP(*p*) where *p* is the corresponding timestamp precision.
Timestamp with time zone
CURRENT TIMESTAMP(*p*) WITH TIME ZONE where *p* is the corresponding timestamp precision.

A default value other than the one that is listed above can be specified in one of the following forms:

constant
Specifies a constant as the default value for the column. The value of the constant must conform to the rules for assigning that value to the column. A hexadecimal graphic string constant (GX) cannot be specified.

SESSION_USER or USER
Specifies the value of the SESSION_USER (USER) special register at the time of an insert or update operation or LOAD as the default value for the column. The data type of the column must be a character string with a length attribute greater than or equal to the length attribute of the SESSION_USER special register.

CURRENT SQLID
Specifies the value of the SQL authorization ID of the process at the time of an SQL data change statement or LOAD as the default value for the column. The data type of the column must be a character string with a length attribute greater than or equal to the length attribute of the CURRENT SQLID special register.

NULL
Specifies null as the default value for the column. If NOT NULL was specified, DEFAULT NULL must not be specified within the same *column-definition*.

GENERATED
Specifies that DB2 generates values for the column. GENERATED must be specified if the column is to be considered an IDENTITY column. If DEFAULT is specified for the column for an update operation, DB2 generates a value for both GENERATED ALWAYS and GENERATED BY DEFAULT.

ALWAYS
Specifies that DB2 always generates a value for the column when a row is inserted into the table.

BY DEFAULT
Specifies that DB2 generates a value for the column when a row is inserted into the table unless a value is specified. BY DEFAULT is the recommended value only when you are using data propagation.

Defining a column as GENERATED BY DEFAULT does not necessarily guarantee the uniqueness of the values. To ensure uniqueness of the values, define a unique, single-column index on the column.

AS IDENTITY
Specifies that the column is an identity column for the table. A table can have only one identity column. AS IDENTITY can be specified only if the data type for the column is an exact numeric type with a scale of zero (SMALLINT, INTEGER, BIGINT, DECIMAL with a scale of zero).

An identity column is implicitly NOT NULL. An identity column cannot have a DEFAULT clause. For the descriptions of the identity attributes, see the description of the AS IDENTITY clause in "CREATE TABLE" on page 1388.

NOT NULL
Specifies that the column cannot contain nulls. Omission of NOT NULL indicates that the column can contain nulls.

LIKE *table-name* **or** *view-name*
Specifies that the columns of the table have the same name, data type, and nullability attributes as the columns of the identified table or view. If a table is identified, the column default attributes are also defined by that table. If row permissions or column access control is enforced for the table specified by *table-name*, row and column access controls are not inherited by the new table. The name specified must identify a table, view, synonym, or alias that exists at the current server. The identified table must not be an auxiliary table.

The privilege set must include the SELECT privilege on the identified table or view.

This clause is similar to the LIKE clause on CREATE TABLE, but it has the following differences:

- If LIKE results in a column having a LOB data type, a ROWID data type, or distinct type, the DECLARE GLOBAL TEMPORARY TABLE statement fails.
- In addition to these data type restrictions, if any column has any other attribute value that is not allowed in a declared temporary table, that attribute value is ignored. The corresponding column in the new temporary table has the default value for that attribute unless otherwise indicated.

table-name or *view-name* must not contain a Unicode column in an EBCDIC table

When the identified object is a table, the column name, data type, nullability, and default attributes are determined from the columns of the specified table; any identity column attributes are inherited only if the INCLUDING IDENTITY COLUMN ATTRIBUTES clause is specified.

as-result-table
Specifies that the table definition is based on the column definitions from the result of a query expression.

The behavior of these column attributes is controlled with the INCLUDING or USING TYPE DEFAULTS clauses, which are defined below.

AS *(fullselect)*
Specifies an implicit definition of *n* columns for the declared global temporary table, where *n* is the number of columns that would result from the *fullselect*. The columns of the new table are defined by the columns that result from the *fullselect*. Every select list element must have a unique name. The AS clause can be used in the select-clause to provide unique names. The implicit definition includes the column name, data type, and nullability characteristic of each of the result columns of *fullselect*. A column of the new table that corresponds to an implicitly hidden column of a base table referenced in the *fullselect* is not considered hidden in the new table.Row and column access controls that are enforced on the base table are not cascaded to the new table.

The result table of the *fullselect* must not contain a column that has a LOB data type, a ROWID data type, an XML data type or a distinct type.

If *fullselect* results in other column attributes that are not applicable for a declared temporary table, those attributes are ignored in the implicit definition for the declared temporary table.

If *fullselect* results in a row change timestamp column, the corresponding column of the new table inherits only the data type of the row change timestamp column. The new column is not considered as a generated column.

The *fullselect* must not refer to host variables or include parameter markers (question marks). The outermost SELECT list of the *fullselect* must not reference data that is encoded with different CCSID sets, and must not result in a column that is an array.

WITH NO DATA
Specifies that the *fullselect* is not executed. You can use the INSERT INTO

statement with the same *fullselect* specified in the AS clause to populate the declared temporary table with the set of rows from the result table of the *fullselect*.

copy-options
Specifies whether identity column attributes and column defaults are inherited from the definition of the source of the result table.

EXCLUDING IDENTITY COLUMN ATTRIBUTES or INCLUDING IDENTITY COLUMN ATTRIBUTES
Specifies whether identity column attributes are inherited from the columns resulting from the *fullselect*, *table-name*, or *view-name*.

EXCLUDING IDENTITY COLUMN ATTRIBUTES
Specifies that the table does not inherit the identity attributes of the columns resulting from the *fullselect*, *table-name*, or *view-name*.

INCLUDING IDENTITY COLUMN ATTRIBUTES
Specifies that the table inherits the identity attributes, if any, of the columns resulting from the *fullselect* or *table-name*. In general, the identity attributes are copied if the element of the corresponding column in the table or *fullselect* is the name of a table column that directly or indirectly maps to the name of a base table column that is an identity column.

If the INCLUDING IDENTITY COLUMN ATTRIBUTES clause is specified with the AS fullselect clause, the columns of the new table do not inherit the identity attribute in the following cases:

- The select list of the *fullselect* includes multiple instances of an identity column name (that is, selecting the same column more than once).
- The select list of the *fullselect* includes multiple identity columns (that is, it involves a join).
- The identity column is included in an expression in the select list.
- The *fullselect* includes a set operation.

If INCLUDING IDENTITY COLUMN ATTRIBUTES is not specified, the new table will not have an identity column.

If the LIKE clause identifies a view, INCLUDING IDENTITY COLUMN ATTRIBUTES must not be specified.

EXCLUDING COLUMN DEFAULTS, INCLUDING COLUMN DEFAULTS, or USING TYPE DEFAULTS
Specifies whether the table inherits the default values of the columns of the fullselect. EXCLUDING COLUMN DEFAULTS, INCLUDING COLUMN DEFAULTS, and USING TYPE DEFAULTS must not be specified if the LIKE clause is specified.

EXCLUDING COLUMN DEFAULTS
Specifies that the table does not inherit the default values of the columns of the fullselect. The default values of the column of the new table are either null or there are no default values. If the column can be null, the default is the null value. If the column cannot be null, there is no default value, and an error occurs if a value is not provided for a column on an insert operation for the new table.

INCLUDING COLUMN DEFAULTS
Specifies that the table inherits the default values of the columns of the fullselect. A default value is the value that is assigned to the column

when a value is not specified on an insert operation or LOAD. Columns resulting from the fullselect that are not updatable will not have a default defined in the corresponding column of the created table.

USING TYPE DEFAULTS
Specifies that the default values for the declared temporary table depend on the data type of the columns that result from *fullselect*, as follows:

Data type
Default value
Numeric
0
Fixed-length character string
Blanks
Fixed-length graphic string
Blanks
Fixed-length binary string
Hexadecimal zeros
Varying-length string
A string of length 0
Date CURRENT DATE
Time CURRENT TIME
Timestamp
CURRENT TIMESTAMP(*p*) where *p* is the corresponding timestamp precision.
Timestamp(*integer*) with time zone
CURRENT TIMESTAMP(*p*) WITH TIME ZONE where *p* is the corresponding timestamp precision.

CCSID *encoding-scheme*
Specifies the encoding scheme for string data that is stored in the table. For declared temporary tables, the encoding scheme for the data cannot be specified for the table space or database, and all data in one table space or the database need not use the same encoding scheme. Because there can be only one work file database for all declared temporary tables for each DB2 member, there can be a mixture of encoding schemes in both the database and each table space.

For the creation of temporary tables, the CCSID clause can be specified whether or not the LIKE clause is specified. If the CCSID clause is specified, the encoding scheme of the new table is the scheme that is specified in the CCSID clause. If the CCSID clause is not specified, the encoding scheme of the new table is the same as the scheme for the table specified in the LIKE clause or as the scheme for the table identified by the AS (fullselect) clause.

ASCII Specifies that the data is encoded by using the ASCII CCSIDs of the server.

EBCDIC
Specifies that the data is encoded by using the EBCDIC CCSIDs of the server.

UNICODE
Specifies that the data is encoded by using the UNICODE CCSIDs of the server.

An error occurs if the CCSIDs for the encoding scheme have not been defined. Usually, each encoding scheme requires only a single CCSID. Additional CCSIDs are needed when mixed, graphic, or UNICODE data is used.

ON COMMIT
Specifies what happens to the table for a commit operation. The default is ON COMMIT DELETE ROWS.

DELETE ROWS
Specifies that all of the rows of the table are deleted if there is no open cursor that is defined as WITH HOLD that references the table.

PRESERVE ROWS
Specifies that all of the rows of the table are preserved. Thread reuse capability is not available to any application process or thread that contains, at its most recent commit, an active declared temporary table that was defined with the ON COMMIT PRESERVE ROWS clause.

DROP TABLE
Specifies that the table is implicitly dropped at commit if there is no open cursor that is defined as WITH HOLD that references the table. If there is an open cursor defined as WITH HOLD on the table at commit, the rows are preserved.

LOGGED or NOT LOGGED
Specifies whether operations for the table are to be logged. This option also applies to any indexes that are associated with the table. Indexes inherit the logging attribute from their associated tables.

LOGGED
Specifies that insert, update, or delete operations for the declared temporary table are logged. Create and drop actions for the table are also logged. This is the default option.

NOT LOGGED
Specifies that insert, update, or delete operations for the declared temporary table are not logged. However, create and drop actions for the table are logged.

ON ROLLBACK DELETE ROWS
Specifies that when a ROLLBACK or ROLLBACK TO SAVEPOINT statement is issued, all rows of the global temporary table are deleted. This is the default.

ON ROLLBACK PRESERVE ROWS
Specifies that when a ROLLBACK or ROLLBACK TO SAVEPOINT statement is issued, all rows of the global temporary table are preserved.

If a ROLLBACK or ROLLBACK TO SAVEPOINT statement is issued, the following actions occur for tables that were created or dropped:

- If the table was created within the unit of work or savepoint, the table is dropped.
- If the table was dropped within the unit of work or savepoint, the table is re-created without any data.

For statements that insert multiple rows, the ATOMIC and NOT ATOMIC CONTINUE ON SQLEXCEPTION options of the INSERT statement determine the result of an error. If ATOMIC is specified, an error during insertion causes all rows in the global temporary table to be deleted. If NOT ATOMIC CONTINUE ON SQLEXCEPTION is

specified, an error during insertion causes all rows in the table to be deleted, but the next insert is processed. At the end of the insert, the table includes only the rows that were inserted after the last error.

Restriction: In CREATE TABLESPACE and ALTER TABLESPACE statements, LOG YES and LOG NO can be used as syntax alternatives for LOGGED and NOT LOGGED, respectively. These syntax alternatives cannot be used in a DECLARE GLOBAL TEMPORARY TABLE statement.

Notes

Instantiation, scope, and termination: For the following explanations, P denotes an application process, and T is a declared temporary table executed in P:

- An empty instance of T is created when a DECLARE GLOBAL TEMPORARY TABLE statement is executed in P.
- Any SQL statement in P can reference T, and any of those references to T in P is a reference to that same instance of T. ()

  If a DECLARE GLOBAL TEMPORARY statement is specified within an SQL PL compound statement, the scope of the declared temporary table is the application process and not just the compound statement. A declared temporary table cannot be defined multiple times by the same name in other compound statements in that application process, unless the table has been dropped explicitly.
- If T was declared at a remote server, the reference to T must use the same DB2 connection that was used to declare T and that connection must not have been terminated after T was declared. When the connection to the database server at which T was declared terminates, T is dropped.
- If T was defined with the ON COMMIT DELETE ROWS clause specified implicitly or explicitly, when a commit operation terminates a unit of work in P and there is no open WITH HOLD cursor in P that is dependent on T, the commit deletes all rows from T.
- If T is defined with the ON COMMIT DROP TABLE clause, when a commit operation terminates a unit of work in P and no program in P has a WITH HOLD cursor open that is dependent on T, the commit includes the operation DROP TABLE T.
- When a rollback operation terminates a unit of work or savepoint in P, and that unit of work or savepoint includes the declaration of SESSION.T, the changes to table T are undone.

  When a rollback operation terminates a unit of work or savepoint in P, and that unit of work or savepoint includes the declaration of SESSION.T, the rollback drops table T.

  When a rollback operation terminates a unit of work or savepoint in P, and that unit of work or savepoint includes the drop of the declaration of declared temporary table SESSION.T, the rollback undoes the drop of table T.
- When the application process that declared T terminates, T is dropped.
- When a rollback operation terminates a unit of work or a savepoint in P, and that unit of work or savepoint includes a modification to SESSION.T the following actions occur:
  - If NOT LOGGED was specified, all rows from SESSION.T are deleted unless ON ROLLBACK PRESERVE ROWS was also specified.
  - If NOT LOGGED was not specified, the changes to table T are undone.

- If NOT LOGGED was specified and an INSERT, UPDATE or DELETE statement fails during execution (not a compilation error), all rows from SESSION.T are deleted.
- When a rollback operation terminates a unit of work or a savepoint in P, and that unit of work or savepoint includes the declaration of SESSION.T, the rollback includes the operation DROP SESSION.T.
- When a rollback operation terminates a unit of work or a savepoint in P, and that unit of work or savepoint includes the drop of a declared temporary table SESSION.T, the rollback undoes the drop of the table. If NOT LOGGED was specified, the table is also emptied.
- When the application process that declared T terminates or disconnects from the database, T is dropped and its instantiated rows are destroyed.

Privileges: When a declared temporary table is defined, PUBLIC is implicitly granted all table privileges on the table and authority to drop the table. These implicit privileges are not recorded in the DB2 catalog and cannot be revoked. This enables any SQL statement in the application process to reference a declared temporary table that has already been defined in that application process.

Referring to a declared temporary table in other SQL statements: Many SQL statements support declared temporary tables. To refer to a declared temporary table in an SQL statement other than DECLARE GLOBAL TEMPORARY TABLE, you must qualify the table name with SESSION. You can either specify SESSION explicitly in the table name or use the QUALIFIER bind option to specify SESSION as the qualifier for all SQL statements in the plan or package.

If you use SESSION as the qualifier for a table name but the application process does not include a DECLARE GLOBAL TEMPORARY TABLE statement for the table name, DB2 assumes that you are not referring to a declared temporary table. DB2 resolves such table references to a table whose definition is persistent and appears in the DB2 catalog tables.

With the exception of the DECLARE GLOBAL TEMPORARY TABLE statement, any static SQL statement that references a declared temporary table is incrementally bound at run time. This is because the definition of the declared temporary table does not exist until the DECLARE GLOBAL TEMPORARY statement is executed in the application process that contains those SQL statements and the definition does not persist when the application process finishes running.

When a plan or package is bound, any static SQL statement (other than the DECLARE GLOBAL TEMPORARY TABLE statement) that references a *table-name* that is qualified by SESSION, regardless of whether the reference is for a declared temporary table, is not completely bound. However, the bind of the plan or package succeeds if there are no other errors. These static SQL statements are then incrementally bound at run time when the static SQL statement is issued. Object dependencies are not recorded in SYSIBM.SYSPLANDEP or SYSIBM.SYSPACKDEP tables. These incremental binds are necessary because:

- The definition of the declared temporary table does not exist until the DECLARE GLOBAL TEMPORARY TABLE statement for the table is executed in the same application process that contains those SQL statements. Therefore, DB2 must wait until the plan or package is run to determine if SESSION.*table-name* refers to a base table or a declared temporary table.
- The definition of a declared temporary table does not persist after the table it is explicitly dropped (DROP statement), implicitly dropped (ON COMMIT DROP TABLE), or the application process that defined it finishes running. When the

application process terminates or is re-used as a reusable application thread, the instantiated rows of the table are deleted and the definition of the declared temporary table is dropped if it has not already been explicitly or implicitly dropped.

After the plan or package is bound, any static SQL statement that refers to a *table-name* that is qualified by SESSION has a new statement status of M in the DB2 catalog table (STATUS column of SYSIBM.SYSSTMT or SYSIBM.SYSPACKSTMT).

Thread reuse: If a declared temporary table is defined in an application process that is running as a local thread, the application process or local thread that declared the table qualifies for explicit thread reuse if:

- The table was defined with both the default ON COMMIT DELETE ROWS attribute and the NOT LOGGED ON ROLLBACK DELETE ROWS attribute.
- The table was defined with PRESERVE ROWS specified on either the ON COMMIT or NOT LOGGED ON ROLLBACK option and the table was explicitly dropped with the DROP TABLE statement before the thread's commit operation.
- The table was defined with the ON COMMIT DROP TABLE attribute. When a declared temporary table is defined with the ON COMMIT DROP TABLE and a commit occurs, the table is implicitly dropped if there are no open cursors defined with the WITH HOLD option.

When the thread is reused, the declared temporary table is dropped and its rows are destroyed. However, if you do not explicitly or implicitly drop all declared temporary tables before or when your thread performs a commit and the thread becomes idle waiting to be reused, as with all thread reuse situations, the idle thread holds resources and locks. This includes some declared temporary table resources and locks on the table spaces and the database descriptor (DBD) for the work file database. So, instead of using the implicit drop feature of thread reuse to drop your declared temporary tables, it is recommended that you:

- Use the DROP TABLE statement to explicitly drop your declared temporary tables before the thread performs a commit and becomes idle.
- Define the declared temporary tables with ON COMMIT DROP TABLE clause so that the tables are implicitly dropped when a commit occurs.

Explicitly dropping the tables before a commit occurs or having them implicitly dropped when the commit occurs enables you to maximize the use of declared temporary table resources and release locks when multiple threads are using declared temporary table.

Remote threads qualify for thread reuse differently than local threads. If a declared temporary table is defined (with or without ON COMMIT DELETE ROWS) in an application process that is running as a remote or DDF thread (also known as Database Access Thread or DBAT), the remote thread qualifies for thread reuse only when the declared temporary table is explicitly dropped before the thread performs a commit operation. Dropping the declared temporary table enables the remote thread to qualify for the implicit thread reuse that is supported for DDF threads via connection pooling and to become an inactive DBAT (type 1 inactive thread) or an inactive connection (type 2 inactive thread).

Parallelism support: Only I/O and CP parallelism are supported. Any query that involves a declared temporary table is limited to parallel tasks on a single CPC.

Restrictions on the use of declared temporary tables: Declared temporary tables cannot:

- Be specified in referential constraints.
- Be referenced in any SQL statements that are defined in an SQL function body (CREATE FUNCTION or ALTER FUNCTION statements), a trigger body (CREATE TRIGGER statement). If you refer a table name that is qualified with SESSION in a trigger body, DB2 assumes that you are referring to a base table.
- Be referenced in a CREATE INDEX statement unless the schema name of the index is SESSION.

In addition, do not refer to a declared temporary table in any of the following statements.

ALTER INDEX
ALTER TABLE
COMMENT
CREATE ALIAS
CREATE FUNCTION (TABLE LIKE clause)
CREATE PROCEDURE (TABLE LIKE clause)
CREATE TRIGGER
CREATE VIEW
GRANT (table or view privileges)
LABEL
LOCK TABLE
REFRESH TABLE
RENAME
REVOKE (table or view privileges)

Declared global temporary tables and dynamic statement caching: The DB2 dynamic statement cache feature does not support dynamic SQL statements that reference declared temporary tables, even if the SQL statement also includes references to base or persistent tables. DB2 will not insert such statements into the dynamic statement cache. Instead, these dynamic statements are processed as if statement caching is not in effect. Declared temporary tables are unique and specific to an application process or DB2 thread, cannot be shared across threads, are not described in the DB2 catalog, and do not persist beyond termination of the DB2 thread or application process. These attributes prevent the use of the dynamic statement cache feature where tables and SQL statements are shared across threads or application processes.

Table space requirements in the work file database: DB2 stores all declared temporary tables in the work file database. You cannot define a declared temporary table unless a table space with at least an 32KB page size exists in the work file database.

Alternative syntax and synonyms: To provide compatibility with previous releases, DB2 allows you to specify:

- LONG VARCHAR as a synonym for VARCHAR(*integer*) and LONG VARGRAPHIC as a synonym for VARGRAPHIC(*integer*) when defining the data type of a column.

  However, the use of these synonyms is not encouraged because after the statement is processed, DB2 considers a LONG VARCHAR column to be VARCHAR and a LONG VARGRAPHIC column to be VARGRAPHIC.
- DEFINITION ONLY as a synonym for WITH NO DATA.
- TIMEZONE can be specified as an alternative to TIME ZONE.

Examples

Example 1: Define a declared temporary table with column definitions for an employee number, salary, commission, and bonus.

```
DECLARE GLOBAL TEMPORARY TABLE SESSION.TEMP_EMP
   (EMPNO     CHAR(6)   NOT NULL,
    SALARY    DECIMAL(9, 2),
    BONUS     DECIMAL(9, 2),
    COMM      DECIMAL(9, 2))
    CCSID EBCDIC
    ON COMMIT PRESERVE ROWS;
```

Example 2: Assume that base table USER1.EMPTAB exists and that it contains three columns, one of which is an identity column. Declare a temporary table that has the same column names and attributes (including identity attributes) as the base table.

```
DECLARE GLOBAL TEMPORARY TABLE TEMPTAB1
   LIKE USER1.EMPTAB
   INCLUDING IDENTITY
   ON COMMIT PRESERVE ROWS;
```

In the above example, DB2 uses SESSION as the implicit qualifier for TEMPTAB1.

DECLARE STATEMENT

The DECLARE STATEMENT statement is used for application program documentation. It declares names that are used to identify prepared SQL statements.

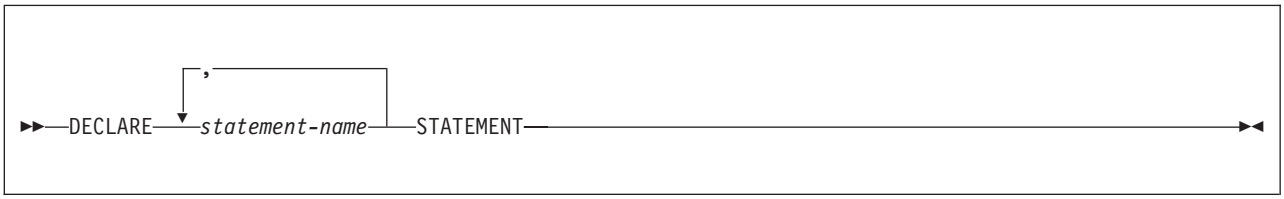
Invocation

This statement can only be embedded in an application program. It is not an executable statement.

Authorization

None required.

Syntax



Description

statement-name **STATEMENT**
Lists one or more names that are used in your application program to identify prepared SQL statements.

Example

This example shows the use of the DECLARE STATEMENT statement in a PL/I program.

```
EXEC SQL DECLARE OBJECT_STATEMENT STATEMENT;

EXEC SQL INCLUDE SQLDA;
EXEC SQL DECLARE C1 CURSOR FOR OBJECT_STATEMENT;

( SOURCE_STATEMENT IS "SELECT DEPTNO, DEPTNAME,
  MGRNO FROM DSN8B10.DEPT WHERE ADMRDEPT = 'A00'"  )

EXEC SQL PREPARE OBJECT_STATEMENT FROM SOURCE_STATEMENT;
EXEC SQL DESCRIBE OBJECT_STATEMENT INTO SQLDA;

/* Examine SQLDA */

EXEC SQL OPEN C1;

DO WHILE (SQLCODE = 0);
  EXEC SQL FETCH C1 USING DESCRIPTOR SQLDA;

/* Print results */

END;

EXEC SQL CLOSE C1;
```

DECLARE TABLE

The DECLARE TABLE statement is used for application program documentation. It also provides the precompiler with information used to check your embedded SQL statements. (The DCLGEN subcommand can be used to generate declarations for tables and views described in any accessible DB2 catalog.

For more on DCLGEN, see *DB2 Application Programming and SQL Guide* and *DB2 Command Reference*.)

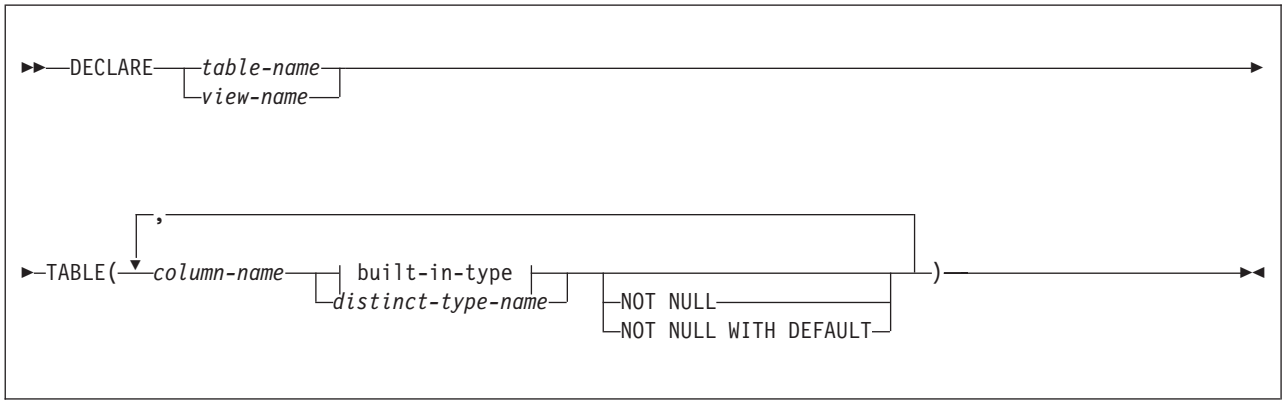
Invocation

This statement can only be embedded in an application program. It is not an executable statement.

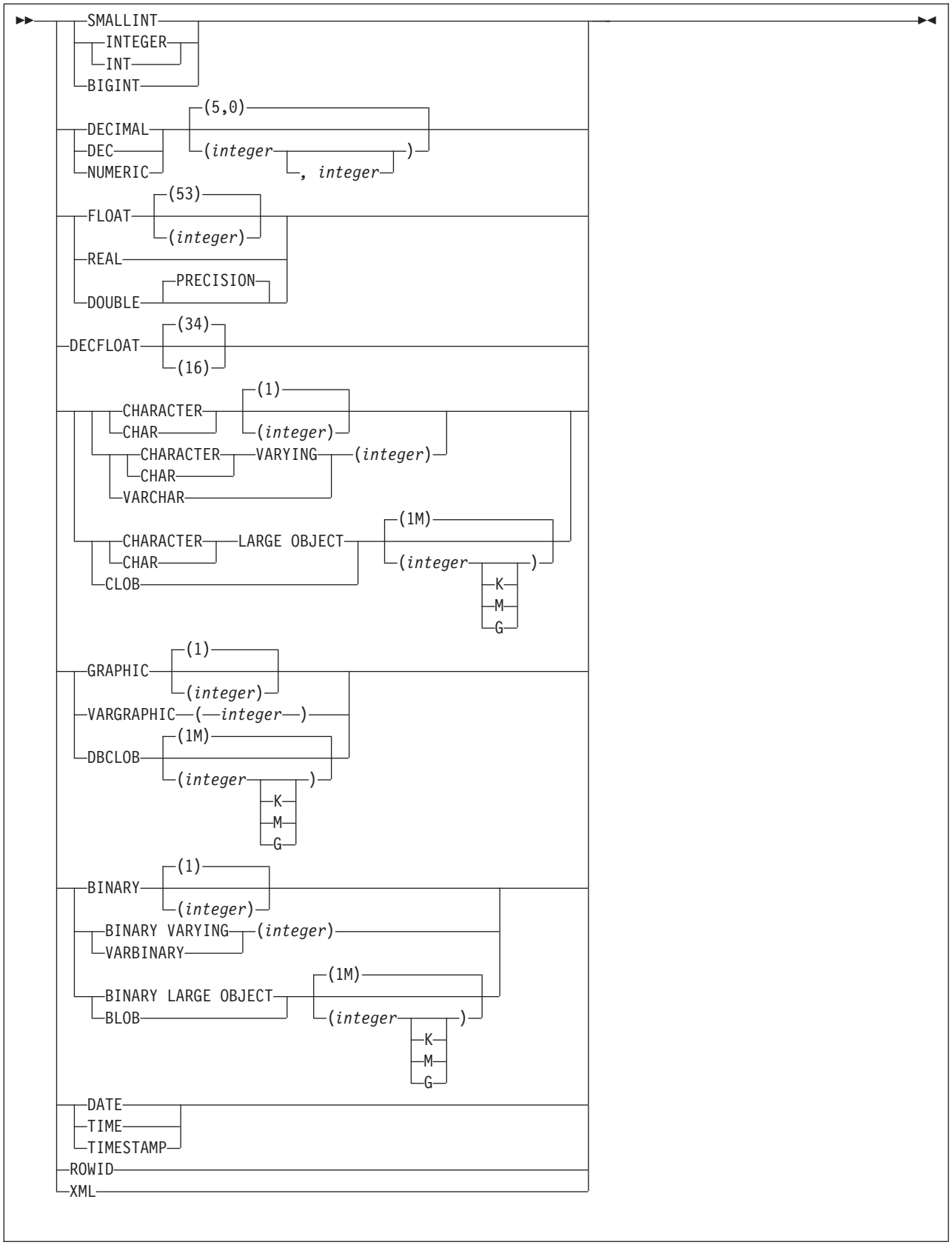
Authorization

None required.

Syntax



built-in-type:


SMALLINT
INTEGER
INT
BIGINT
DECIMAL
DEC
NUMERIC
(5,0)
(integer)
(integer, integer)
FLOAT
(53)
(integer)
REAL
DOUBLE
PRECISION
DECFLOAT
(34)
(16)
CHARACTER
CHAR
(1)
(integer)
CHARACTER VARYING
CHAR VARYING
VARCHAR
(integer)
CHARACTER LARGE OBJECT
CHAR LARGE OBJECT
CLOB
(1M)
(integer)
K
M
G
GRAPHIC
(1)
(integer)
VARGRAPHIC
(integer)
DBCLOB
(1M)
(integer)
K
M
G
BINARY
(1)
(integer)
BINARY VARYING
VARBINARY
(integer)
BINARY LARGE OBJECT
BLOB
(1M)
(integer)
K
M
G
DATE
TIME
TIMESTAMP
ROWID
XML

Description

table-name **or** *view-name*
Specifies the name of the table or view to document. If the table is defined in your application program, the description of the table in the SQL statement in which it is defined (for example, CREATE TABLE or DECLARE GLOBAL TEMPORARY TABLE statement) and the DECLARE TABLE statement must be identical.

column-name
Specifies the name of a column of the table or view.

The precompiler uses these names to check for consistency of names within your SQL statements. It also uses the data type to check for consistency of names and data types within your SQL statements.

built-in-type
Specifies the built-in data type of the column. Use one of the built-in data types.

SMALLINT
For a small integer.

INTEGER or INT
For a large integer.

BIGINT
For a big integer.

DECIMAL(*integer***,***integer***) or DEC(***integer***,***integer***)**
DECIMAL(*integer***) or DEC(***integer***)**
DECIMAL or DEC
For a decimal number. The first integer is the precision of the number. That is, the total number of digits, which can range from 1 to 31. The second integer is the scale of the number. That is, the number of digits to the right of the decimal point, which can range from 0 to the precision of the number.

You can use DECIMAL(*p*) for DECIMAL(*p*,0) and DECIMAL for DECIMAL(5,0).

You can also use the word NUMERIC instead of DECIMAL. For example, NUMERIC(8) is equivalent to DECIMAL(8). Unlike DECIMAL, NUMERIC has no allowable abbreviation.

FLOAT(*integer***)**
FLOAT
For a floating-point number. If *integer* is between 1 and 21 inclusive, the format is single precision floating-point. If the integer is between 22 and 53 inclusive, the format is double precision floating-point.

You can use DOUBLE PRECISION or FLOAT for FLOAT(53).

REAL
For single precision floating-point.

DOUBLE or DOUBLE PRECISION
For double precision floating-point

DECFLOAT(*integer***)**
For a decimal floating-point number. The value of *integer* must be either 16

or 34 and represents the number of significant digits that can be stored. If *integer* is omitted, the DECFLOAT column will be capable of representing 34 significant digits.

CHARACTER(*integer***) or CHAR(***integer***)**
CHARACTER or CHAR
For a fixed-length character string of length *integer*, which can range from 1 to 255. If the length specification is omitted, a length of 1 character is assumed.

VARCHAR(*integer***), CHAR VARYING(***integer***), or CHARACTER VARYING(***integer***)**
For a varying-length character string of maximum length *integer*, which can range from 1 to the maximum record size minus 10 bytes. See Table 119 on page 1448 to determine the maximum record size.

CLOB(*integer* **[K|M|G]), CHAR LARGE OBJECT(***integer* **[K|M|G]), or CHARACTER LARGE OBJECT(***integer* **[K|M|G])**
CLOB, CHAR LARGE OBJECT, or CHARACTER LARGE OBJECT
For a character large object (CLOB) string of the specified maximum length in bytes. The maximum length must be in the range of 1 to 2 147 483 647. A CLOB column has a varying-length. It cannot be referenced in certain contexts regardless of its maximum length. For more information, see "Restrictions using LOBs" on page 97.

When *integer* is not specified, the default length is **1M**. The maximum value that can be specified for *integer* depends on whether a units indicator is also specified as shown in the following list.

integer The maximum value for *integer* is 2 147 483 647. The maximum length of the string is *integer*.

integer **K**
The maximum value for *integer* is 2 097 152. The maximum length is 1024 times *integer*.

integer **M**
The maximum value for *integer* is 2048. The maximum length is 1 048 576 times *integer*.

integer **G**
The maximum value for *integer* is 2. The maximum length is 1 073 741 824 times *integer*.

If you specify a value that evaluates to 2 gigabytes (2 147 483 648), DB2 uses a value that is one byte less, or 2 147 483 647.

GRAPHIC(*integer***)**
GRAPHIC
For a fixed-length graphic string of length *integer*, which can range from 1 to 127. If the length specification is omitted, a length of 1 character is assumed.

VARGRAPHIC(*integer***)**
For a varying-length graphic string of maximum length *integer*, which must range from 1 to $n/2$, where n is the maximum row size minus 2 bytes.

CCSID 1200
Specifies that the column is a Unicode column encoded in UTF16. This clause must not be specified for an ASCII or Unicode table.

DBCLOB(*integer* **[K|M|G])**

DBCLOB
For a double-byte character large object (DBCLOB) string of the specified maximum length in double-byte characters. The maximum length must be in the range of 1 through 1 073 741 823. A DBCLOB column has a varying-length. It cannot be referenced in certain contexts regardless of its maximum length. For more information, see "Restrictions using LOBs" on page 97.

When *integer* is not specified, the default length is **1M**. The meaning of *integer* K|M|G is similar to CLOB. The difference is that the number specified is the number of double-byte characters.

BINARY(*integer*)
A fixed-length binary string of length *integer*. The *integer* can range from 1 through 255. If the length specification is omitted, a length of 1 byte is assumed.

BINARY VARYING(*integer*) or VARBINARY(*integer*)
A varying-length binary string of maximum length *integer*, which can range from 1 through 32704. The length is limited by the page size of the table space.

BLOB (*integer* [K|M|G] or BINARY LARGE OBJECT(*integer* [K|M|G])
BLOB or BINARY LARGE OBJECT
For a binary large object (BLOB) string of the specified maximum length in bytes. The maximum length must be in the range of 1 through 2 147 483 647. A BLOB column has a varying-length. It cannot be referenced in certain contexts regardless of its maximum length. For more information, see "Restrictions using LOBs" on page 97.

When *integer* is not specified, the default length is **1M**. The meaning of *integer* K|M|G is the same as for CLOB.

DATE
For a date.

TIME
For a time.

TIMESTAMP(*integer*) WITHOUT TIME ZONE
For a timestamp. *integer* specifies the optional timestamp precision attribute and must be in the range from 0 to 12. The timestamp precision denotes the number of fractional second digits that are included in the timestamp. The default is 6.

TIMESTAMP(*integer*) WITH TIME ZONE
For a timestamp with time zone. *integer* specifies the optional timestamp precision attribute and must be in the range from 0 to 12. The timestamp precision denotes the number of fractional second digits that are included in the timestamp. The default is 6.

ROWID
For a row ID type.

A table can have only one ROWID column. The values in a ROWID column are unique for every row in the table and cannot be updated. You must specify NOT NULL with ROWID.

XML
For an XML document. Only well-formed XML documents can be inserted into an XML column.

If the XML column is the first XML column that you create for the table, a BIGINT DOCID column is implicitly created and is used to store a unique document identifier for the XML columns of a row.

distinct-type-name
Specifies the distinct type (user-defined data type) of the column. An implicit or explicit schema name qualifies the name.

NOT NULL
Specifies that the column does not allow null values and does not provide a default value.

NOT NULL WITH DEFAULT
Specifies that the column does not allow null values but provides a default value.

Notes

Error handling during processing: If an error occurs during the processing of the DECLARE TABLE statement, a warning message is issued, and the precompiler continues processing your source program.

Documenting a distinct type column: Although you can specify the name of a distinct type as the data type of a column in the DECLARE TABLE statement, use the built-in data type on which the distinct type is based instead. Using the base type enables the precompiler to check the embedded SQL statements for errors; otherwise, error checking is deferred until bind time.

To determine the source data type of the distinct type, check the value of column SOURCETYPE in catalog table SYSDATATYPES.

Examples

Example 1: Declare the sample employee table, DSN8B10.EMP.

```
EXEC SQL DECLARE DSN8B10.EMP TABLE
  (EMPNO     CHAR(6)     NOT NULL,
   FIRSTNME  VARCHAR(12) NOT NULL,
   MIDINIT   CHAR(1)     NOT NULL,
   LASTNAME  VARCHAR(15) NOT NULL,
   WORKDEPT  CHAR(3)             ,
   PHONENO   CHAR(4)             ,
   HIREDATE  DATE                ,
   JOB       CHAR(8)             ,
   EDLEVEL   SMALLINT            ,
   SEX       CHAR(1)             ,
   BIRTHDATE DATE                ,
   SALARY    DECIMAL(9,2)        ,
   BONUS     DECIMAL(9,2)        ,
   COMM      DECIMAL(9,2)        );
```

Example 2: Assume that table CANADIAN_SALES keeps information for your company's sales in Canada. The table was created with the following definition:

```
CREATE TABLE CANADIAN_SALES
  (PRODUCT_ITEM   INTEGER,
   MONTH          INTEGER,
   YEAR           INTEGER,
   TOTAL          CANADIAN_DOLLAR);
```

CANADIAN_DOLLAR is a distinct type that was created with the following statement:

```
CREATE TYPE CANADIAN_DOLLAR AS DECIMAL(9,2);
```

Declare the CANADIAN_SALES table, using the source type for CANADIAN_DOLLAR instead of the distinct type name.

```
DECLARE TABLE CANADIAN_SALES
  (PRODUCT_ITEM   INTEGER,
   MONTH          INTEGER,
   YEAR           INTEGER,
   TOTAL          DECIMAL(9,2);
```

DECLARE VARIABLE

The DECLARE VARIABLE statement defines a CCSID for a host variable and the subtype of the variable. When it appears in an application program, the DECLARE VARIABLE statement causes the DB2 precompiler to tag a host variable with a specific CCSID. When the host variable appears in an SQL statement, the DB2 precompiler places this CCSID into the structures that it generates for the SQL statement.

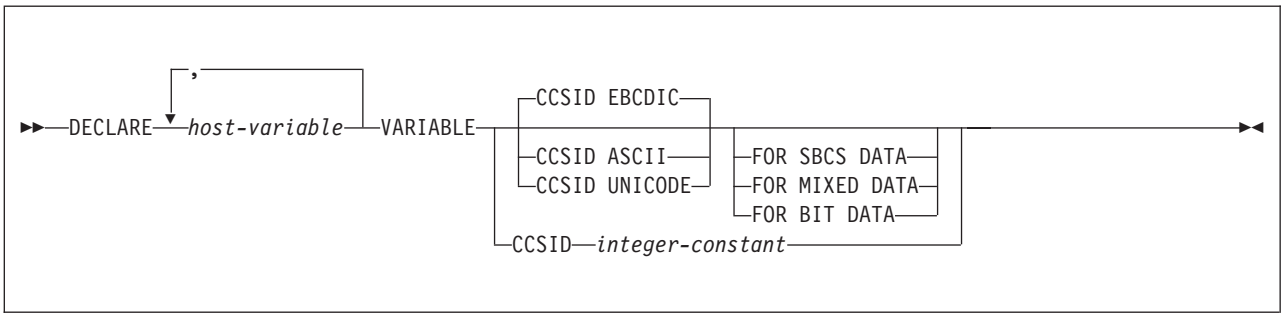
Invocation

This statement can only be embedded in an application program. It is not an executable statement.

Authorization

None required.

Syntax



Description

host-variable
Identifies a character or graphic string host variable defined in the program. An indicator variable cannot be specified for the host-variable.

CCSID ASCII, EBCDIC, or UNICODE
Specifies that the appropriate default CCSID for the specified encoding scheme of the server should be used.

CCSID ASCII
Specifies that the default ASCII CCSID for the type of the variable at the server should be used.

CCSID EBCDIC
Specifies that the default EBCDIC CCSID for the type of the variable at the server should be used. CCSID EBCDIC is the default if this option is not specified.

CCSID UNICODE
Specifies that the default UNICODE CCSID for the type of the variable at the server should be used.

FOR SBCS DATA, FOR MIXED DATA, or FOR BIT DATA
Specifies the type of data contained in the variable *host-variable*. The FOR clause cannot be specified when declaring a graphic host variable.

For ASCII or EBCDIC data, if this clause is not specified when declaring a character host variable, the default is FOR SBCS DATA if MIXED DATA = NO on the installation panel DSNTIPF. The default is FOR MIXED DATA if MIXED DATA = YES on the installation panel DSNTIPF.

For UNICODE data, the default is always FOR MIXED DATA, regardless of the setting of MIXED DATA on the installation panel DSNTIPF.

FOR SBCS DATA
Specifies that the values of the host variable can contain only SBCS (single-byte character set) data.

FOR MIXED DATA
Specifies that the values of the host variable can contain both SBCS data and DBCS data.

FOR BIT DATA
Specifies that the values of the host-variable are not associated with a coded character set and, therefore, are never converted. The CCSID of a FOR BIT DATA host variable is 65535.

CCSID *integer-constant*
Specifies that the values of the host variable contain data that is encoded using CCSID *integer-constant*. If the integer is an SBCS CCSID, the host variable is SBCS data. If the integer is a mixed data CCSID, the host variable is mixed data. For character host variables, the CCSID specified must be an SBCS, mixed CCSID, or UNICODE (UTF-8) CCSID. For graphic host variables, the CCSID specified must be a DBCS or UNICODE (UTF-16) CCSID. The valid range of values for the integer is 1 - 65533.

Notes

Placement of statement: The DECLARE VARIABLE statement can be specified anywhere in an application program that SQL statements are valid with the following exception. The DECLARE VARIABLE statement must occur before an SQL statement that refers to a host variable specified in the DECLARE VARIABLE statement.

CCSID exceptions for EXECUTE IMMEDIATE or PREPARE: When the host variable appears in an SQL statement, the DB2 precompiler places the appropriate numeric CCSID into the structures it generates for the SQL statement. This placement of the CCSID occurs for any SQL statement other than the EXECUTE IMMEDIATE or PREPARE statements. The placement of the CCSID also occurs for a *host-variable* in an EXECUTE IMMEDIATE or PREPARE statement, but it does not occur for a variable in a *string-expression* in an EXECUTE IMMEDIATE or PREPARE statement.

If a PL/1 application program contains at least one DECLARE VARIABLE statement, a *string-expression* in any EXECUTE IMMEDIATE or PREPARE statement cannot be preceded by a colon. An expression that consists of just a variable name preceded by a colon is interpreted as a *host-variable*.

Specific host languages: If a DECLARE VARIABLE statement is used in an assembler source program, the ONEPASS SQL processing option must not be used. If a DECLARE VARIABLE statement is used in a C, C++, or PL/I source program, the TWOPASS SQL processing option must be used. For those languages, or COBOL, the host-variable definition can either precede or follow a DECLARE VARIABLE statement that refers to that variable. If a DECLARE VARIABLE

statement is used in a FORTRAN source program, then the host-variable definition must precede the DECLARE VARIABLE statement.

Example

Example: Define the following host variables using PL/I data types: FRED as fixed length bit data, JEAN as fixed length UTF-8 (mixed) data, DAVE as varying length UTF-8 (mixed) data, PETE as fixed length graphic UTF-16 data, and AMBER as varying length graphic UTF-16 data.

Use the DECLARE VARIABLE statement to specify a data subtype or CCSID for these host variables: FRED as CCSID EBCDIC, JEAN as CCSID 1208 or CCSID UNICODE, DAVE as CCSID 1208 or CCSID UNICODE, PETE as CCSID 1200 or CCSID UNICODE, and AMBER as CCSID 1200 or CCSID UNICODE.

```
EXEC SQL BEGIN DECLARE SECTION;
   DCL FRED CHAR(10);
       EXEC SQL DECLARE :FRED VARIABLE CCSID EBCDIC FOR BIT DATA;
   DCL JEAN CHAR(30);
       EXEC SQL DECLARE :JEAN VARIABLE CCSID 1208;
   DCL DAVE CHAR(9) VARYING;
       EXEC SQL DECLARE :DAVE VARIABLE CCSID UNICODE;
   DCL PETE GRAPHIC(10);
       EXEC SQL DECLARE :PETE VARIABLE CCSID 1200;
   DCL AMBER GRAPHIC(20) VARYING;
       EXEC SQL DECLARE :AMBER VARIABLE CCSID UNICODE;
EXEC SQL END DECLARE SECTION;
```

DELETE

The DELETE statement deletes rows from a table or view or activates an instead of delete trigger. The table or view can be at the current server or any DB2 subsystem with which the current server can establish a connection. Deleting a row from a view deletes the row from the table on which the view is based if no instead of trigger is defined for the delete operation on this view. If such a trigger is defined, the trigger is activated instead of the delete operation.

There are two forms of this statement:

- The *searched* DELETE form is used to delete one or more rows, optionally determined by a search condition.
- The *positioned* DELETE form specifies that one or more rows corresponding to the current cursor position are to be deleted.

Invocation

This statement can be embedded in an application program or issued interactively. A positioned DELETE is embedded in an application program. Both the embedded and interactive forms are executable statements that can be dynamically prepared.

Authorization

Authority requirements depend on whether the object identified in the statement is a user-defined table, a catalog table, or a view, and whether the statement is a searched DELETE and SQL standard rules are in effect:

When a table other than a catalog table is identified: The privilege set must include at least one of the following:

- The DELETE privilege on the table
- Ownership of the table
- DBADM authority on the database that contains the table
- SYSADM authority

If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

When a catalog table is identified: The privilege set must include at least one of the following:

- DBADM authority on the catalog database
- SYSCTRL authority
- SYSADM authority

When a view is identified: The privilege set must include at least one of the following:

- The DELETE privilege on the view
- SYSADM authority

If the *search-condition* in a searched DELETE contains a reference to a column of the table or view, or the *expression* in the *assignment-clause* contains a reference to a column of the table or view, the privilege set must include at least one of the following:

- The SELECT privilege on the table or view

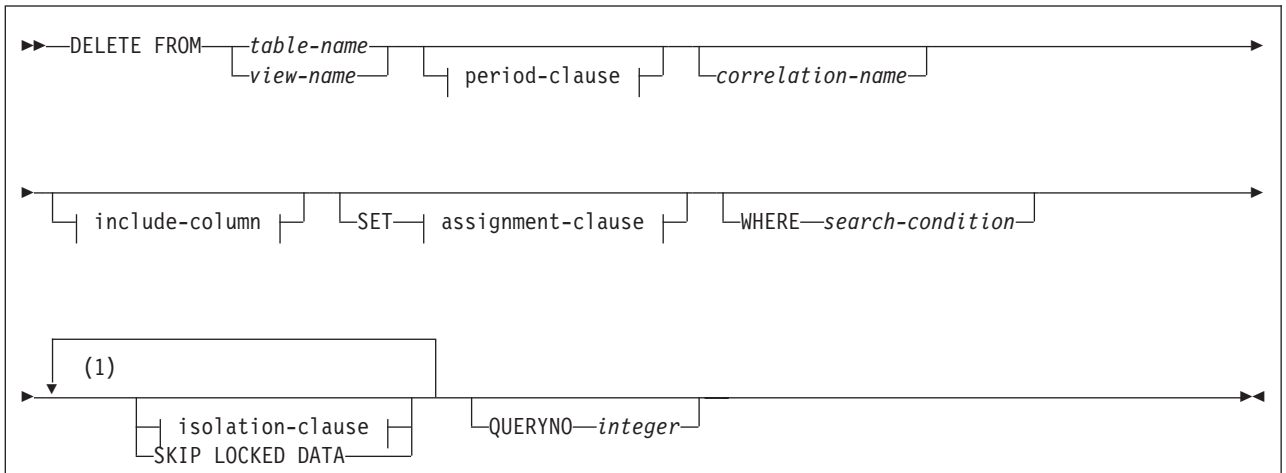
- Ownership of the table or view
- DBADM authority on the database that contains the table, if the target is a table and that table that is not a catalog table
- DATAACCESS
- SYSADM authority

If the *search-condition* in a searched DELETE includes a subquery, or if the *assignment-clause* includes a *scalar-fullselect* or a *row-fullselect*, see "Authorization" on page 762 for an explanation of the authorization required.

The owner of a view, unlike the owner of a table, might not have DELETE authority on the view (or might have DELETE authority without being able to grant it to others). The nature of the view itself can preclude its use for DELETE. For more information, see the description of authority in "CREATE VIEW" on page 1527.

If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set is determined by the DYNAMICRULES behavior in effect (run, bind, define, or invoke) and is summarized in Table 94 on page 841. (For more information on these behaviors, including a list of the DYNAMICRULES bind option values that determine them, see "Authorization IDs and dynamic SQL" on page 75.)

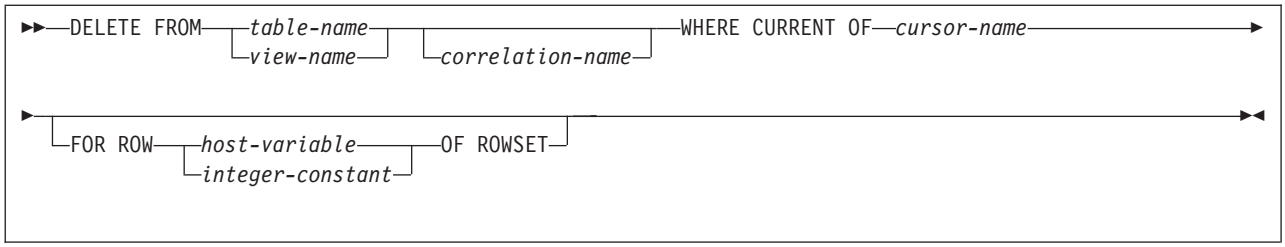
searched delete:



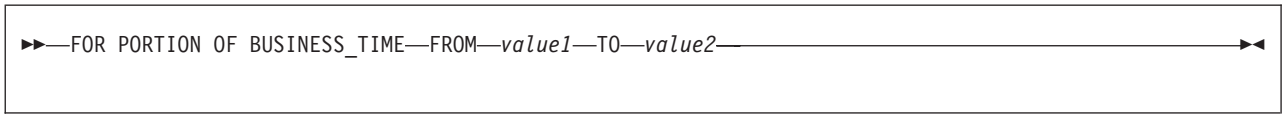
Notes:

1 The same clause must not be specified more than one time.

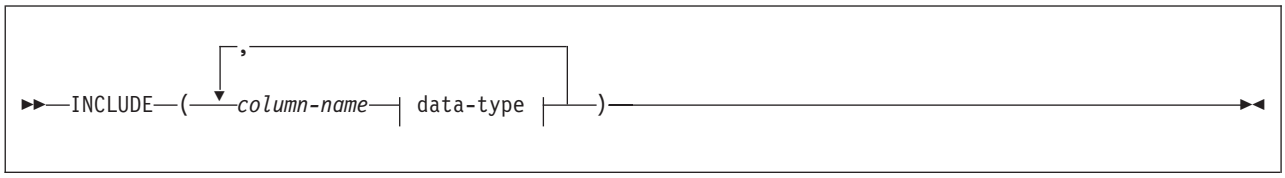
positioned delete:



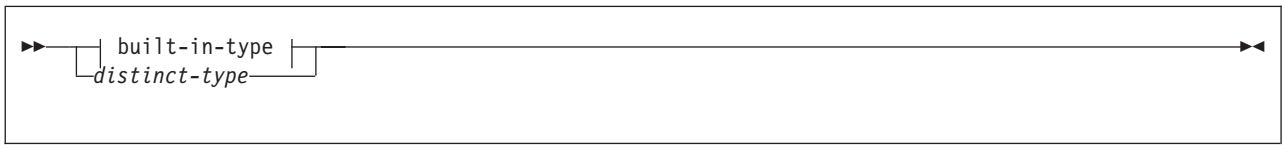
period-clause:



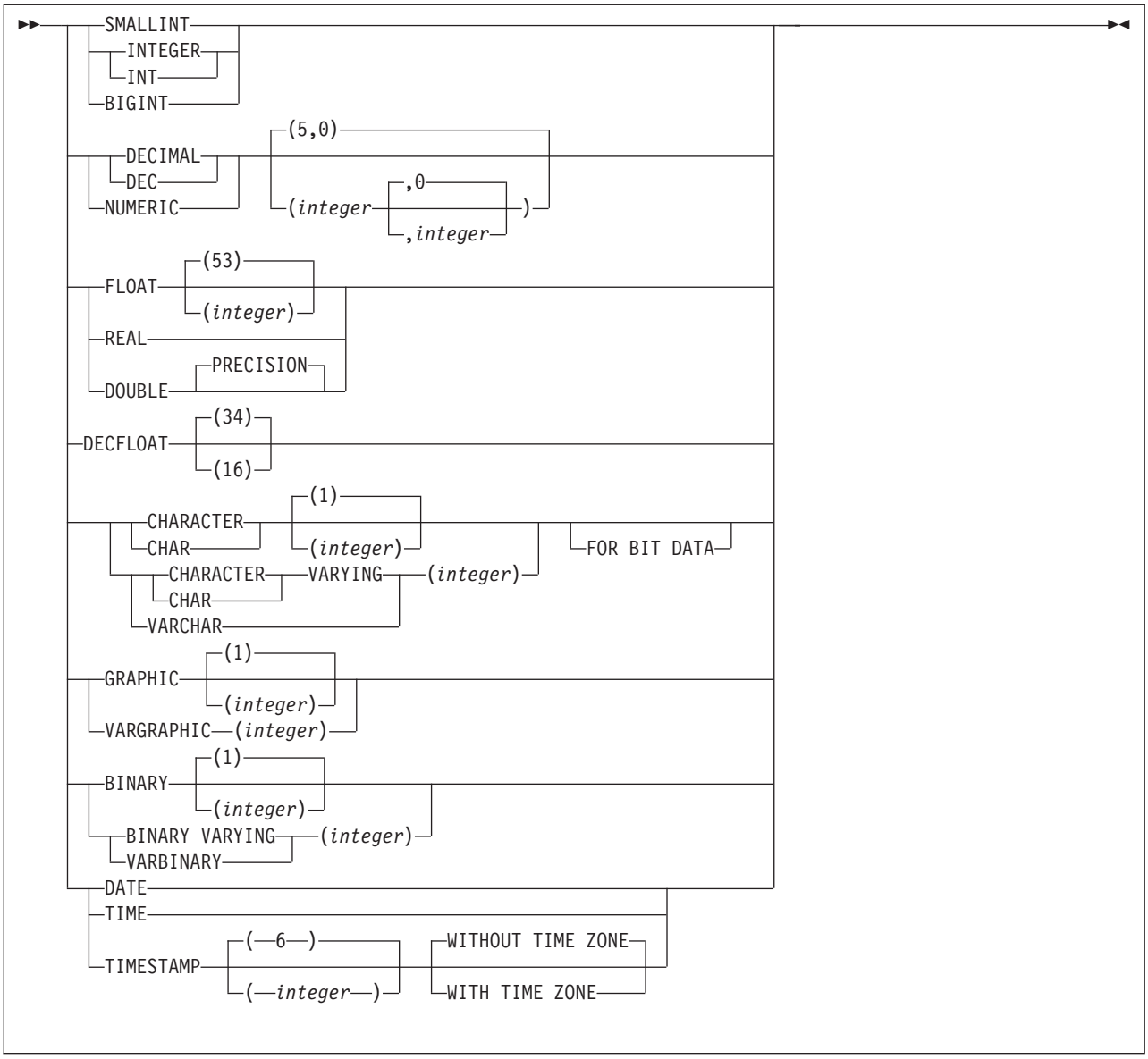
include-column:



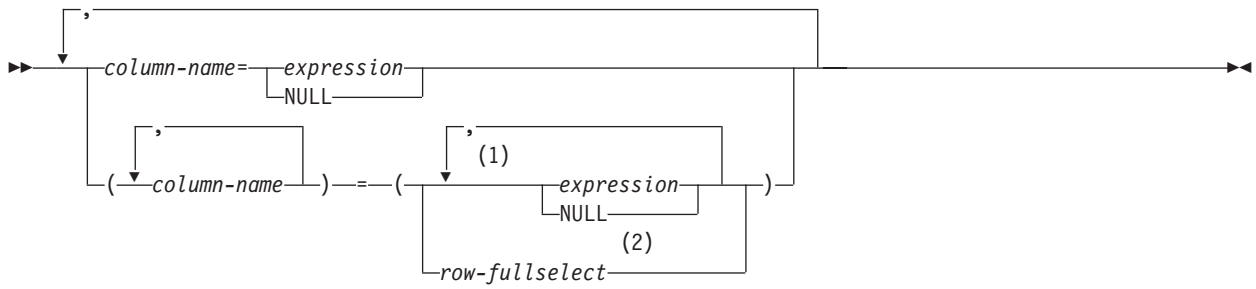
data-type:



built-in-type:



assignment clause:

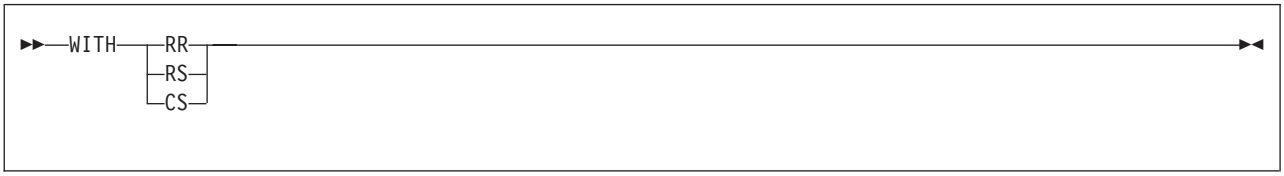


Notes:

1 The number of *expressions* and NULL keywords must match the number of *column-names*.

2 The number of columns in the select list must match the number of *column-names*.

isolation-clause:



Description

FROM *table-name* **or** *view-name*

Identifies the table or view from which rows are to be deleted. The name must identify a table or view that exists at the DB2 subsystem that is identified by the implicitly or explicitly specified location name. The name must not identify:

- An auxiliary table
- A catalog table for which deletes are not allowed
- A view of such a catalog table
- A read-only view (For a description of a read-only view, see "CREATE VIEW" on page 1527.)
- A system-maintained materialized query table
- A table that is implicitly created for an XML column
- An archive-enabled table if the SYSIBMADM.GET_ARCHIVE global variable is set to Y, the ARCHIVESENSITIVE bind option is set to YES, and the operation is a positioned delete

In an IMS or CICS application, the DB2 subsystem that contains the identified table or view must be a remote server that supports two-phase commit.

period-clause

Specifies that a period clause applies to the target of the delete operation. The same period name must not be specified more than one time. If the target of the delete operation is a view:

- The FROM clause of the outer fullselect of the view definition must include a reference, directly or indirectly, to an application-period temporal table.
- An INSTEAD OF trigger must not be defined for that view.

FOR PORTION OF BUSINESS_TIME
Specifies that the delete only applies to row values for the portion of the BUSINESS_TIME period in the row that is specified by the period clause. BUSINESS_TIME must be a period that is defined on the table.

FROM *value1* **TO** *value2*
Specifies that the delete operation applies to rows for the period that is specified from *value1* to *value2*. No rows are deleted if *value1* is greater than or equal to *value2*, or if *value1* or *value2* is the null value.

For the period condition that is specified with **FROM** *value1* **TO** *value2*, the period that is specified by *period-name* in a row of the target table of the delete covers one of the following ranges:

- If the value of the begin column is less than *value1* and the value of the end column is greater than *value1*, the range overlaps the beginning of the specified period.
- If the value of the end column is greater than or equal to *value2* and the value of the begin column is less than *value2*, the range overlaps the end of the specified period.
- If the value for the begin column for *period-name* in the row is greater than or equal to *value1* and the value for the corresponding end column in the row is less than or equal to *value2*, the range is fully contained within the specified period.
- If the row overlaps the beginning of the specified period or the end of the specified period, but not both, the range is partially contained in the specified period.
- If the period in the row overlaps the beginning of the specified period and overlaps the end of the specified period, the range fully overlaps the specified period.
- If both columns of *period-name* are less than or equal to *value1* or are greater than or equal to *value2*, the range is not contained in the period.

If the period *period-name* in a row is not contained in the specified period, the row is not deleted. Otherwise, the delete operation is applied based on the specification of the PORTION OF clause and how the values in the columns of *period-name* overlap the specified period as follows:

- If the period *period-name* in a row is fully contained within the specified period, the row is deleted.
- If the period *period-name* in a row is partially contained in the specified period and overlaps the beginning of the specified period:
  - The row is deleted.
  - A row is inserted using the original values from the row, except that the end column is set to *value1*.
- If the period *period-name* in a row is partially contained in the specified period and overlaps the end of the specified period:
  - The row is deleted.
  - A row is inserted using the original values from the row, except that the begin column is set to *value2*.
- If the period *period-name* in a row fully overlaps the specified period:
  - The row is deleted.
  - A row is inserted using the original values from the row, except that the end column is set to *value1*.

– An additional row is inserted using the original values from the row, except that the begin column is set to *value*2.

Any existing delete triggers are activated for the rows that are deleted, and any existing insert triggers are activated for the rows that are implicitly inserted.

value1, *value2*
Specifies expressions that return a value of a built-in data type. The result of each expression must be comparable to the data type of the columns of the specified period. See the comparison rules described in "Assignment and comparison" on page 121. Each expression can contain any of the following supported operands:

- A constant
- A special register
- A variable (host variable, SQL variable, SQL parameter, or transition variable)
- An array element specification
- A built-in scalar function whose arguments are supported operands
- A CAST specification where the cast operand is a supported operand
- An expression that uses arithmetic operators and operands

Each expression must not have a timestamp precision that is greater than the precision of the columns for the period.

If the begin and end columns of the period are defined as TIMESTAMP WITHOUT TIME ZONE, each expression must not return a value of a timestamp with a time zone.

A period clause for a view must not contain a global variable or an untyped parameter marker.

correlation-name
Specifies an alternate name that can be used within the *search-condition* to designate the table or view. (For an explanation of correlation names, see "Correlation names" on page 209.)

include-column
Specifies a set of columns that are included, along with the columns of *table-name* or *view-name*, in the result table of the DELETE statement when it is nested in the FROM clause of the outer fullselect that is used in a subselect, SELECT statement, or in a SELECT INTO statement. The included columns are appended to the end of the list of columns that is identified by *table-name* or *view-name*. If no value is assigned to a column that is specified by an *include-column*, a NULL value is returned for that column.

INCLUDE
Introduces a list of columns that are to be included in the result table of the DELETE statement. The included columns are only available if the DELETE statement is nested in the FROM clause of a SELECT statement or a SELECT INTO statement.

column-name
Specifies the name for a column of the result table of the DELETE statement that is not the same name as another included column nor a column in the table or view that is specified in *table-name* or *view-name*.

data-type
Specifies the data type of the included column. The included columns are nullable.

built-in-type
Specifies a built-in data type. See "CREATE TABLE" on page 1388 for a description of each built-in type.

The CCSID 1208 and CCSID 1200 clauses must not be specified for an include column.

distinct-type
Specifies a distinct type. Any length, precision, or scale attributes for the column are those of the source type of the distinct type as specified by using the CREATE TYPE statement.

SET
Introduces the assignment of values to columns.

assignment-clause
The assignment-clause introduces a list of one or more *column-names* and the values that are to be assigned to the columns. The *column-names* are the only columns that can be set using the *assignment-clause*.

column-name
Identifies an INCLUDE column.

Assignments to included columns are only processed when the DELETE statement is nested in the FROM clause of a SELECT statement or a SELECT INTO statement. The columns that are named in the INCLUDE clause are the only columns that can be set using the SET clause. The null value is returned for an included column that is not set by using an explicit SET clause.

expression
Indicates the new value of the column. The *expression* is any expression of the type described in "Expressions" on page 240. It must not include an aggregate function.

A *column-name* in an expression must identify a column of the table or view. For each row that is deleted, the value of the column in the expression is the value of the column in the row before the row is deleted.

NULL
Specifies the null value as the new value of the column. Specify NULL only for nullable columns.

row-fullselect
Specifies a fullselect that returns a single row. The column values are assigned to each of the corresponding *column-names*. If the fullselect returns no rows, the null value is assigned to each column; an error occurs if any column that is to be deleted is not nullable. An error also occurs if there is more than one row in the result.

If the fullselect refers to columns that are to be deleted, the value of such a column in the fullselect is the value of the column in the row before the row is deleted.

WHERE
Specifies the rows to be deleted. You can omit the clause, give a search condition, or specify a cursor. For a created temporary table or a view of a

created temporary table, you must omit the clause. When the clause is omitted, all the rows of the table or view are deleted.

search-condition
Is any search condition as described in Chapter 2, "Language elements," on page 53. Each *column-name* in the search condition, other than in a subquery, must identify a column of the table or view.

The search condition is applied to each row of the table or view and the deleted rows are those for which the result of the search condition is true.

If the search condition contains a subquery, the subquery can be thought of as being executed each time the search condition is applied to a row, and the results used in applying the search condition. In actuality, a subquery with no correlated references is executed just once, whereas it is possible that a subquery with a correlated reference must be executed once for each row.

Let T2 denote the object table of a DELETE statement and let T1 denote a table that is referred to in the FROM clause of a subquery of that statement. T1 must not be a table that can be affected by the DELETE on T2. Thus, the following rules apply:

- T1 must not be a dependent of T2 in a relationship with a delete rule of CASCADE or SET NULL, unless the result of the subquery is materialized before the DELETE action is executed.
- T1 must not be a dependent of T3 in a relationship with a delete rule of CASCADE or SET NULL if deletes of T2 cascade to T3.

WHERE CURRENT OF *cursor-name*
Identifies the cursor to be used in the delete operation. *cursor-name* must identify a declared cursor as explained in the description of the DECLARE CURSOR statement in "DECLARE CURSOR" on page 1535. If the DELETE statement is embedded in a program, the DECLARE CURSOR statement must include *select-statement* rather than *statement-name*.

The table or view named must also be named in the FROM clause of the SELECT statement of the cursor, and the result table of the cursor must be capable of being deleted. For an explanation of read-only result tables, see Read-only cursors. Note that the object of the DELETE statement must not be identified as the object of the subquery in the WHERE clause of the SELECT statement of the cursor.

If the cursor is ambiguous and the plan or package was bound with CURRENTDATA(NO), DB2 might return an error to the application if DELETE WHERE CURRENT OF is attempted for any of the following:

- A cursor that is using block fetching
- A cursor that is using query parallelism
- A cursor that is positioned on a row that has been modified by this or another application process

When the DELETE statement is executed, the cursor must be open and positioned on a row or rowset of the result table.

- If the cursor is positioned on a single row, that row is the one deleted, and after the deletion the cursor is positioned before the next row of its result table. If there is no next row, the cursor positioned after the last row.

- If the cursor is positioned on a rowset, all rows corresponding to the rows of the current rowset are deleted, and after the deletion the cursor is positioned before the next rowset of its result table. If there is no next rowset, the cursor positioned after the last rowset.

A positioned DELETE must not be specified for a cursor that references a view on which an instead of delete trigger is defined, even if the view is an updatable view.

FOR ROW n OF ROWSET
Specifies which row of the current rowset is to be deleted. The corresponding row of the rowset is deleted, and the cursor remains positioned on the current rowset. If the rowset consists of a single row, or all other rows in the rowset have already been deleted, then the cursor is positioned before the next rowset of the result table. If there is no next rowset, the cursor is positioned after the last rowset.

host-variable or *integer-constant* is assigned to an integral value *k*. If *host-variable* is specified, it must be an exact numeric type with scale zero, must not include an indicator variable, and *k* must be in the range of 1 to 32767. The cursor must be positioned on a rowset, and the specified value must be a valid value for the set of rows most recently retrieved for the cursor.

If the specified row cannot be deleted, an error is returned. It is possible that the specified row is within the bounds of the rowset most recently requested, but the current rowset contains less than the number of rows that were implicitly or explicitly requested when that rowset was established.

If, via a positioned delete against a sensitive static cursor that specifies a particular row of the current rowset, and that row has been identified as a delete hole (that is, a row in the result table whose corresponding row has deleted from the base table), an error is returned.

If, via a positioned delete against a sensitive static cursor that specifies a particular row of the current rowset, and that row has been identified as an update hole (that is, a row in the result table whose corresponding row has been updated so that it no longer satisfies a predicate of the SELECT statement), an error is returned.

It is possible for another application process to delete a row in the base table of the SELECT statement so that the specified row of the cursor no longer has a corresponding row in the base table. An attempt to delete such a row results in an error.

If the FOR ROW n OF ROWSET clause is not specified, the current position of cursor determines the rows that are affected by the statement:

- If the cursor is positioned on a single row, that row is the one deleted. After the row is deleted, the cursor is positioned before the next row of its result table. If there is no next row, the cursor positioned after the last row.
- If the cursor is positioned on a rowset, all rows corresponding to the rows of the current rowset are deleted. After the rows are deleted, the cursor is positioned before the next rowset of its result table. If there is no next rowset, the cursor positioned after the last rowset.

isolation-clause
Specifies the isolation level used when locating the rows to be deleted by the statement.

WITH
Introduces the isolation level, which may be one of the following:

RR Repeatable read
RS Read stability
CS Cursor stability

The default isolation level of the statement is the isolation level of the package or plan in which the statement is bound, with the package isolation taking precedence over the plan isolation. When a package isolation is not specified, the plan isolation is the default.

SKIP LOCKED DATA
The SKIP LOCKED DATA clause specifies that rows are skipped when incompatible locks are held on the row by other transactions. These rows can belong to any accessed table that is specified in the statement. SKIP LOCKED DATA can be used only when isolation CS or RS is in effect and applies only to row level or page level locks.

For DELETE statements, SKIP LOCKED DATA can be specified only in the searched form of the DELETE statement. SKIP LOCKED DATA is ignored if it is specified when the isolation level that is in effect is repeatable read (WITH RR) or uncommitted read (WITH UR). The default isolation level of the statement depends on the isolation level of the package or plan with which the statement is bound, with the package isolation taking precedence over the plan isolation. When a package isolation is not specified, the plan isolation is the default.

QUERYNO *integer*
Specifies the number to be used for this SQL statement in EXPLAIN output and trace records. The number is used for the QUERYNO column of the plan table for the rows that contain information about this SQL statement. This number is also used in the QUERYNO column of the SYSIBM.SYSSTMT and SYSIBM.SYSPACKSTMT catalog tables.

If the clause is omitted, the number associated with the SQL statement is the statement number assigned during precompilation. Thus, if the application program is changed and then precompiled, that statement number might change.

Using the QUERYNO clause to assign unique numbers to the SQL statements in a program is helpful:
- For simplifying the use of optimization hints for access path selection
- For correlating SQL statement text with EXPLAIN output in the plan table

For information on using optimization hints, such as enabling the system for optimization hints and setting valid hint values, and for information on accessing the plan table, see *DB2 Performance Monitoring and Tuning Guide*.

Notes

Delete operation errors:
If an error occurs during the execution of any delete operation, no changes are made. If an error occurs during the execution of a positioned delete, the position of the cursor is unchanged. However, it is possible for an error to make the position of the cursor invalid, in which case the cursor is closed. It is also possible for a delete operation to cause a rollback, in which case the cursor is closed.

Position of cursor:
If an application process deletes a row on which any of its cursors are positioned, those cursors are positioned before the next row of the result table. Let C be a cursor that is positioned before row R (as a result of an

OPEN, a DELETE through C, a DELETE through some other cursor, or a searched DELETE). In the presence of an SQL data change statements that affect the base table from which R is derived, the next FETCH operation referencing C does not necessarily position C on R. For example, the operation can position C on R', where R' is a new row that is now the next row of the result table.

Locking:

Unless appropriate locks already exist, one or more exclusive locks are acquired during the execution of a successful delete operation. Until the locks are released by a commit or rollback operation, the effect of the delete operation can only be perceived by the application process that performed the deletion and the locks can prevent other application processes from performing operations on the table. Locks are not acquired when rows are deleted from a declared temporary table unless all the rows are deleted (DELETE FROM T). When all the rows are deleted from a declared temporary table, a segmented table lock is acquired on the pages for the table and no other table in the table space is affected.

Triggers:

Delete operations can cause triggers to be activated. A trigger might cause other statements to be executed or might raise error conditions that are based on the deleted rows. If a DELETE statement on a view causes an INSTEAD OF trigger to be activated, referential integrity is checked against the updates that are performed in the trigger and not against the underlying tables of the view that cause the trigger to be activated.

Triggers defined on a table for which row or column access control is also enforced:

Row permissions and column masks are not applied to the initial values of transition variables and transition tables. Row and column access control that is enforced for the triggering table is also ignored for any transition variables or transition tables that are referenced in the trigger body or that are passed as arguments to user-defined functions that are invoked in the trigger body. To ensure that no security concern exists for SQL statements in the trigger action (access to sensitive data in transition variables and transition tables, for example), the trigger must be secure. For information about securing a trigger, see “CREATE TRIGGER” on page 1482 and “ALTER TRIGGER” on page 1094.

Referential integrity:

If the identified table or the base table of the identified view is a parent, the rows selected must not have any dependents in a relationship with a delete rule of RESTRICT or NO ACTION. In addition, the delete operation must not cascade to descendent rows that have dependents in a relationship with a delete rule of RESTRICT or NO ACTION.

If the delete operation is not prevented by a RESTRICT or NO ACTION delete rule, the selected rows are deleted and any rows that are dependents of the selected rows are also deleted.

- The nullable columns of foreign keys in any rows that are their dependents in a relationship governed by a delete rule of SET NULL are set to the null value.
- Any rows that are their dependents in a relationship governed by a delete rule of CASCADE are also deleted, and these rules apply, in turn, to those rows.

The only difference between NO ACTION and RESTRICT is when the referential constraint is enforced. RESTRICT (IBM SQL rules) enforces the rule immediately, and NO ACTION (SQL standard rules) enforces the rule at the end of the statement. This difference matters only in the case of a searched DELETE involving a self-referencing constraint that deletes more than one row. NO ACTION might allow the DELETE to be successful where RESTRICT (if it were allowed) would prevent it.

Check constraint:
A check constraint can prevent the deletion of a row in a parent table when there are dependents in a relationship with a delete rule of SET NULL. If deleting a row in the parent table would cause a column in a dependent table to be set to null and there is a check constraint that specifies that the column must not be null, the row is not deleted.

Referential constraints defined on a table for which row or column access control is enforced:
Row and column access controls do not effect referential constraints.

Nesting user-defined functions or stored procedures:
A DELETE statement can implicitly or explicitly refer to user-defined functions or stored procedures. This is known as *nesting* of SQL statements. A user-defined function or stored procedure that is nested within the DELETE must not access the table from which you are deleting rows.

Indexes with VARBINARY columns:
If the identified table has an index on a VARBINARY column or a column that is a distinct type that is based on VARBINARY data type, that index column cannot specify the DESC attribute. To use the SQL data change operation on the identified table, either drop the index or alter the data type of the column to BINARY and then rebuild the index.

Number of rows deleted:
Except as noted below, a delete operation sets SQLERRD(3) in the SQLCA to the number of deleted rows. This number does not include any rows that were deleted as a result of a CASCADE delete rule or a trigger.

DELETE FROM T without a WHERE clause deletes all rows of T. If a table T is contained in a segmented table space and is not a parent table, this deletion will be performed without accessing T. The SQLERRD(3) field is set to -1. (For a complete description of the SQLCA, including exceptions to the above, see "SQL communication area (SQLCA)" on page 2069.

Rules for positioned DELETE with SENSITIVE STATIC scrollable cursor:
When a SENSITIVE STATIC scrollable cursor has been declared, the following rules apply:

- *Delete attempt of delete holes or update holes.* If, with a positioned delete against a SENSITIVE STATIC scrollable cursor, an attempt is made to delete a row that has been identified as a delete hole (that is, a row in the result table whose corresponding row has been deleted from the base table), an error occurs.

  If an attempt is made to delete a row that has been identified as an update hole (that is, a row in the result table whose corresponding row has been updated so that it no longer satisfies the predicate of the SELECT statement), an error occurs.
- *Delete operations.* Positioned delete operations with SENSITIVE STATIC scrollable cursors perform as follows:

1. The SELECT list items in the target row of the base table of the cursor are compared with the values in the corresponding row of the result table (that is, the result table must still agree with the base table). If the values are not identical, the delete operation is rejected and an error occurs. The operation can be attempted again after a successful FETCH SENSITIVE has occurred for the target row.
2. The WHERE clause of the SELECT statement is re-evaluated to determine whether the current values in the base table still satisfy the search criteria. The values in the SELECT list are compared to determine that these values have not changed. If the WHERE clause evaluates as true, and the values in the SELECT list have not changed, the delete operation is allowed to proceed. Otherwise, an error occurs, the delete operation is rejected, and an update hole appears in the cursor.
3. After the base table row is successfully deleted, the temporary result table is updated and the row is marked as a delete hole.

- *Rollback of delete holes.* Delete holes are usually permanent. Once a delete hole is identified, it remains a delete hole until the cursor is closed. However, if a positioned delete using *this* cursor actually caused the creation of the hole (that is, this cursor was used to make the changes that resulted in the hole) and the delete was subsequently rolled back, then the row is no longer considered a delete hole.
- *Result table.* Any deletes, either positioned or searched, to rows of the base table on which a SENSITIVE STATIC scrollable cursor is defined are reflected in the result table if a positioned update or positioned delete is attempted with the scrollable cursor. A SENSITIVE STATIC scrollable cursor sees these deletes when a FETCH SENSITIVE is attempted.

If the FOR ROW *n* OF ROWSET clause is not specified, the entire rowset fetched by the most recent FETCH statement that returned data for the specified cursor is deleted.

Deleting rows from a table with multilevel security:

When you delete rows from a table with multilevel security, DB2 compares the security label of the user (the primary authorization ID) to the security label of the row. The delete proceeds according to the following rules:

- If the security label of the user and the security label of the row are equivalent, the row is deleted.
- If the security label of the user dominates the security label of the row, the user's write-down privilege determines the security the result of the DELETE statement:
  - If the user has write-down privilege or write-down control is not enabled, the row is deleted.
  - If the user does not have write-down privilege and write-down control is enabled, the row is not deleted.
- If the security label of the row dominates the security label of the user, the row is not deleted.

Deleting rows from a table for which row and column access control is enforced:

When a DELETE statement is issued for a table for which row access control is enforced, the rules specified in the row permissions affect whether a row can be deleted. Typically those rules are based on the authorization ID or role of the process.

A table for which row access control is enforced has at least one row permission, the default row permission that prevents all access to the table. When multiple row permissions are defined and enabled for a table, a row access control search condition is derived by using the logical OR operator to the search condition in each enabled permission. This row access control search condition is applied to the table to determine which rows are accessible to the authorization ID or role of the DELETE statement. If the WHERE clause is specified in the DELETE statement, the user-specified predicates are applied on the accessible rows to determine the rows to be deleted. If there is no WHERE clause, the accessible rows are the rows to be deleted.

If there are rows to be deleted, and there is a DELETE trigger for the table, the trigger is activated.

When a DELETE statement is issued for a table for which column access control is enforced, column masks do not affect the DELETE statement.

The preceding rules are not applicable to *include-columns*. *include-columns* are subject to the rules for the select list because they are not the columns of the object table of the DELETE statement.

Other SQL statements in the same unit of work:

The following statements cannot follow a DELETE statement in the same unit of work:

- An ALTER TABLE statement that changes the data type of a column (ALTER COLUMN SET DATA TYPE)
- An ALTER INDEX statement that changes the padding attribute of an index with varying-length columns (PADDED to NOT PADDED or vice versa)

Considerations for a system-period temporal table:

If the DELETE statement has a search condition that contains a correlated subquery that references the history table (explicitly referencing the name of the history table or implicitly referenced through the use of a period specification in the FROM clause), the deleted rows that are stored as historical rows are potentially visible for delete operations for the rows that are subsequently processed for the statement.

The mass delete operation is not used for a DELETE statement that does not contain a search condition if the table is defined as a system-period temporal table.

Considerations for a history table:

When a row of a system-period temporal table is deleted, a historical copy of the row is inserted into the corresponding history table and the end timestamp of the historical row is captured in the form of a system determined value that corresponds to the time of the data change operation. DB2 generates a value using the time-of-day clock during execution of the first data change statement in the transaction that requires a value to be assigned to a row-begin or transaction-start-ID column in a table. This also occurs when a row in a system-period temporal table is deleted. DB2 ensures the uniqueness of the generated values for an end column in a history table across transactions. If a conflicting transaction is updating the same row in the system-period temporal table and the row that is to be inserted into the associated history table will have a value for the end column that is greater than the value of the corresponding begin column, an error is returned.

Effect of the CURRENT TEMPORAL SYSTEM_TIME and CURRENT TEMPORAL BUSINESS_TIME special registers

If the CURRENT TEMPORAL SYSTEM_TIME special register is set to a non-null value, the underlying target of the DELETE statement cannot be a system-period temporal table. This restriction applies regardless of whether the system-period temporal table is directly or indirectly referenced.

If the CURRENT TEMPORAL BUSINESS_TIME special register is set to a non-null value, the DELETE statement is affected if the following conditions are also true:

- An application-period temporal table is the target of the DELETE statement.
- The BUSTIMESENSITIVE bind option is set to YES.

In this situation, DB2 implicitly adds the following additional predicates to the statement:

```
    bt_begin <= CURRENT TEMPORAL BUSINESS_TIME
AND bt_end > CURRENT TEMPORAL BUSINESS_TIME
```

In the preceding code, bt_begin and bt_end are the begin and end columns of the BUSINESS_TIME period of the target table of the DELETE statement.

Deleting rows from archive-enabled tables:

If the target of the DELETE statement is an archive-enabled table, existing rows in the associated archive table are not affected.

When a row of an archive-enabled table is deleted, the effect on the associated archive table is determined by the setting of the SYSIBMADM.MOVE_TO_ARCHIVE global variable. If the global variable is set to Y, a copy of a deleted row is inserted into the associated archive table. Otherwise, a copy of a deleted row is not inserted into the associated archive table.

A data change statement cannot reference an archive-enabled table when a system-period temporal table or application-period temporal table is also referenced.

Syntax alternatives:

For compatibility with other SQL implementations, the FROM keyword can be omitted.

Examples

Assume that the statements in the examples are embedded in PL/I programs.

Example 1: From the table DSN8B10.EMP delete the row on which the cursor C1 is currently positioned.

```
EXEC SQL DELETE FROM DSN8B10.EMP WHERE CURRENT OF C1;
```

Example 2: From the table DSN8B10.EMP, delete all rows for departments E11 and D21.

```
EXEC SQL DELETE FROM DSN8B10.EMP
  WHERE WORKDEPT = 'E11' OR WORKDEPT = 'D21';
```

Example 3: From employee table X, delete the employee who has the most absences.

```
EXEC SQL DELETE FROM EMP X
  WHERE ABSENT = (SELECT MAX(ABSENT) FROM EMP Y
  WHERE X.WORKDEPT = Y.WORKDEPT);
```

Example 4: Assuming that cursor CS1 is positioned on a rowset consisting of 10 rows of table T1, delete all 10 rows in the rowset.

```
EXEC SQL DELETE FROM T1 WHERE CURRENT OF CS1;
```

Example 5: Assuming cursor CS1 is positioned on a rowset consisting of 10 rows of table T1, delete the fourth row of the rowset.

```
EXEC SQL DELETE FROM T1 WHERE CURRENT OF CS1 FOR ROW 4 OF ROWSET;
```

Example 6: Delete rows in table T1 if the value for column COL2 matches the cardinality of array INTA. The array INTA is specified as an argument for the CARDINALITY function in the DELETE statement.

```
CREATE TYPE INTARRAY AS INTEGER ARRAY[6];
DECLARE INTA AS INTARRAY;
SET INTA = ARRAY[1, 2, 3, 4, 5];
CREATE TABLE T1 (COL1 CHAR(7), COL2 INT);
INSERT INTO T1 VALUES('abc', 10);
DELETE FROM T1 WHERE COL2 = CARDINALITY(INTA);
```

DESCRIBE

The DESCRIBE statement obtains information about an object. You can obtain the following types of information with this statement, each of which is described separately.

- Cursors

  Gets information about the result set that is associated with the cursor. This information, such as column information, is put into a descriptor. See "DESCRIBE CURSOR" on page 1591.

- Input parameter markers of a prepared statement.

  Gets information about the input parameter markers in a prepared statement. This information is put into a descriptor. See "DESCRIBE INPUT" on page 1593.

- The output of a prepared statement

  Gets information about a prepared statement or information about the select list columns in a prepared SELECT statement. This information is put into a descriptor. See "DESCRIBE OUTPUT" on page 1596.

- Procedures

  Gets information about the result sets returned by a stored procedure. The information, such as the number of result sets, is put into a descriptor. See "DESCRIBE PROCEDURE" on page 1603.

- Tables

  Gets information about a table or view. This information is put into a descriptor. See "DESCRIBE TABLE" on page 1606.

DESCRIBE CURSOR

The DESCRIBE CURSOR statement obtains information about the result set that is associated with the cursor. The information, such as column information, is put into a descriptor. Use DESCRIBE CURSOR for result set cursors from stored procedures. The cursor must be defined with the ALLOCATE CURSOR statement.

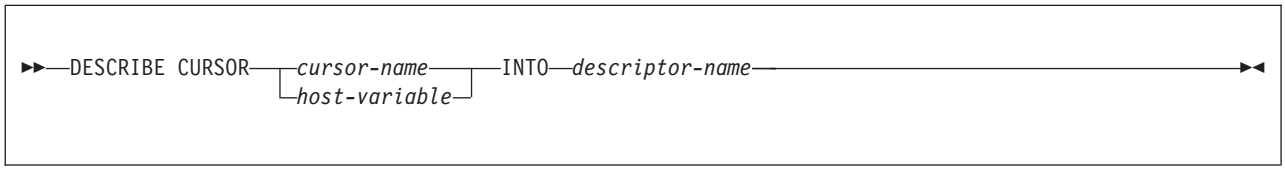
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

None required.

Syntax



Description

cursor-name **or** *host-variable*

Identifies a cursor by the specified *cursor-name* or the cursor name contained in *host-variable*. The name must identify a cursor that has already been allocated in the source program.

A column of the result table of the cursor must not be an array.

If *host-variable* is used:

- It must be a character string variable that has a maximum length of 18 bytes.
- It must not be followed by an indicator variable.
- The cursor name must be left justified within the host variable and must not contain embedded blanks.
- If the length of the cursor name is less than the length of the host variable, it must be padded on the right with blanks.

INTO *descriptor-name*

Identifies an SQL descriptor area (SQLDA). The information returned in the SQLDA describes the columns in the result set associated with the named cursor.

The considerations for allocating and initializing the SQLDA are similar to those of a varying-list SELECT statement. For more information, see *DB2 Application Programming and SQL Guide*.

For REXX: The SQLDA is not allocated before it is used.

After the DESCRIBE CURSOR statement is executed, the contents of the SQLDA are the same as after a DESCRIBE for a SELECT statement, with the following exceptions:

- The first 5 bytes of the SQLDAID field are set to 'SQLRS'.

- Bytes 6 to 8 of the SQLDAID field are reserved. If the cursor is declared WITH HOLD in a stored procedure, the high-order bit of the 8th byte is set to 1.

These exceptions do not apply to a REXX SQLDA, which does not include the SQLDAID field.

Notes

Using cursors for result sets: Column names are included in the information that DESCRIBE CURSOR obtains when the statement that generates the result set is either:

- Dynamic
- Static and the value of field DESCRIBE FOR STATIC on installation panel DSNTIP4 was YES when the package or stored procedure was bound. If the value of the field was NO, the returned information includes only the data type and length of the columns.

Using host variables: If the DESCRIBE CURSOR statement contains host variables, the contents of the host variables are assumed to be in the encoding scheme that was specified in the ENCODING parameter when the package or plan that contains the statement was bound.

Examples

The statements in the following examples are assumed to be in PL/I programs.

Example 1: Place information about the result set associated with cursor C1 into the descriptor named by :sqlda1.

```
EXEC SQL DESCRIBE CURSOR C1 INTO :sqlda1
```

Example 2: Place information about the result set associated with the cursor named by :hv1 into the descriptor named by :sqlda2.

```
EXEC SQL DESCRIBE CURSOR :hv1 INTO :sqlda2
```

DESCRIBE INPUT

The DESCRIBE INPUT statement obtains information about the input parameter markers of a prepared statement.

For an explanation of prepared statements, see "PREPARE" on page 1781 and "DESCRIBE PROCEDURE" on page 1603.

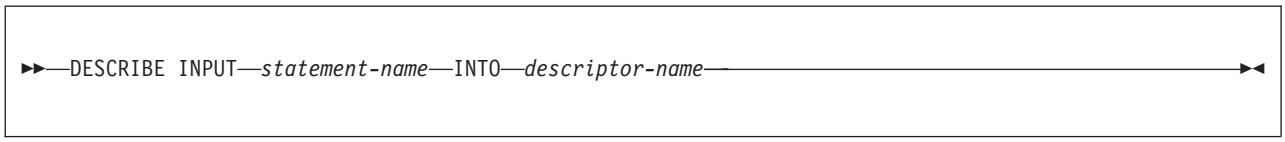
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

The statement can be executed if the privilege set for PREPARE includes the EXPLAIN privilege.

Syntax



Description

statement-name
Identifies the prepared statement. When the DESCRIBE INPUT statement is executed, the name must identify a statement that has been prepared by the application process at the current server. An input parameter marker must not refer to an array value.

INTO *descriptor-name*
Identifies an SQL descriptor area (SQLDA), which is described in "SQL descriptor area (SQLDA)" on page 2079. See "Identifying an SQLDA in C or C++" on page 2099 for how to represent *descriptor-name* in C. The information returned in the SQLDA describes the parameter markers.

Before the DESCRIBE INPUT statement is executed, the user must set the SQLN field in the SQLDA and the SQLDA must be allocated. Considerations for initializing and allocating the SQLDA are similar to those for the DESCRIBE statement (see "DESCRIBE" on page 1590). An occurrence of an extended SQLVAR is needed for each parameter in addition to the required base SQLVAR only if the input data contains LOBs.

For REXX: The SQLDA is not allocated before it is used.

After the DESCRIBE INPUT statement is executed, all the fields in the SQLDA except SQLN are either set by DB2 or ignored. The SQLDA contents are similar to the contents returned for the DESCRIBE statement (see page The SQLDA contents returned after DESCRIBE) with these exceptions:

- In the SQLDAID, DB2 sets the value of the seventh byte only to the space character or '2'. A value of '3' is never used. The value '2' indicates that two SQLVAR entries (an occurrence of both a base SQLVAR and an extended

SQLVAR) are required for each parameter because the input data contains LOBs. The seventh byte is a space character when either of the following conditions is true:
  - The input data does not contain LOBs. Only a base SQLVAR occurrence is needed for each parameter.
  - Only a base SQLVAR occurrence is needed for each column of the result, and the SQLDA is not large enough to contain the returned information.
- The SQLD field is set to the number of parameter markers being described. The value is 0 if the statement being described does not have input parameter markers.
- The SQLNAME field is not used.
- The SQLDATATYPE is set to a nullable, regardless of the usage of the parameter markers in the prepared statement.
- The SQLDATATYPE-NAME is not used if an extended SQLVAR entry is present. DESCRIBE INPUT does not return information about distinct types.

For complete information on the contents of the fields, see "SQL descriptor area (SQLDA)" on page 2079.

Notes

Preparing the SQLDA for OPEN or EXECUTE: This note is relevant if you are applying DESCRIBE INPUT to a prepared statement and you intend to use the SQLDA in an OPEN or EXECUTE statement. To prepare the SQLDA for that purpose:

- Set SQLDATA to a valid address.
- If SQLTYPE is odd, set SQLIND to a valid address.

For the meaning of those fields in that context, see "SQL descriptor area (SQLDA)" on page 2079.

Support for extended dynamic SQL in a distributed environment: Unlike the DESCRIBE statement, which can be used in a distributed environment to describe static SQL statements generated by extended dynamic SQL, you cannot describe host variables in static SQL statements that are generated by extended dynamic SQL. A DESCRIBE INPUT statement issued against such static SQL statements always fails.

For information on how the DESCRIBE statement supports extended dynamic SQL, see Support for extended dynamic SQL in a distributed environment.

Using host variables: If the DESCRIBE INPUT statement contains host variables, the contents of the host variables are assumed to be in the encoding scheme that was specified in the ENCODING parameter when the package or plan that contains the statement was bound.

Example

Execute a DESCRIBE INPUT statement with an SQLDA that has enough SQLVAR occurrences to describe any number of input parameters a prepared statement might have. Assume that five parameter markers at most will need to be described and that the input data does not contain LOBs.

```
     /* STMT1_STR contains INSERT statement with VALUES clause  */
  EXEC SQL  PREPARE STMT1_NAME FROM :STMT1_STR;
 ... /* code to set SQLN to 5 and to allocate the SQLDA         */
```

```
  EXEC SQL  DESCRIBE INPUT STMT1_NAME INTO :SQLDA;
.
.
.
```

This example uses the first technique described in Allocating the SQLDA to allocate the SQLDA.

DESCRIBE OUTPUT

The DESCRIBE OUTPUT statement obtains information about a prepared statement.

For an explanation of prepared statements, see "PREPARE" on page 1781 and "DESCRIBE PROCEDURE" on page 1603.

Invocation

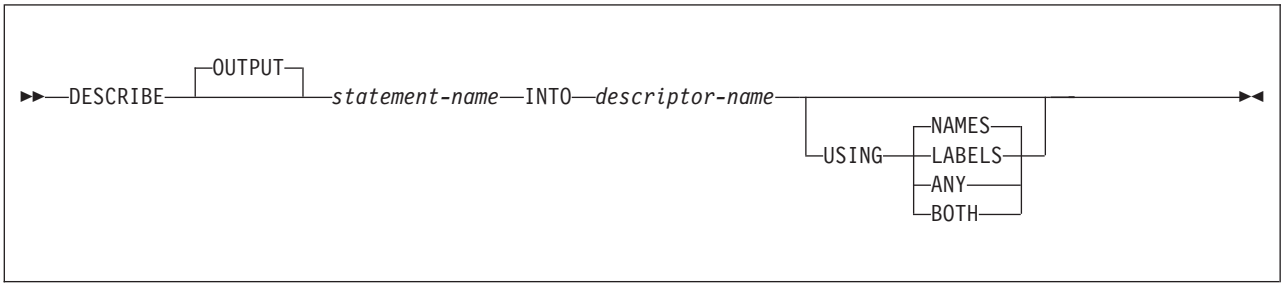
This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

The statement can be executed if the privilege set for PREPARE includes the EXPLAIN privilege.

See "PREPARE" on page 1781 for the authorization required to create a prepared statement.

Syntax



Description

OUTPUT
When a *statement-name* is specified, optional keyword to indicate that the describe will return information about the select list columns in a the prepared SELECT statement.

statement-name
Identifies the prepared statement. When the DESCRIBE statement is executed, the name must identify a statement that has been prepared by the application process at the current server. A column of the result table of the prepared statement must not be an array.

INTO *descriptor-name*
Identifies an SQL descriptor area (SQLDA), which is described in "SQL descriptor area (SQLDA)" on page 2079. See "Identifying an SQLDA in C or C++" on page 2099 for how to represent *descriptor-name* in C.

For languages other than REXX: Before the DESCRIBE statement is executed, the user must set the following variable in the SQLDA and the SQLDA must be allocated.

SQLN Indicates the number of SQLVAR occurrences provided in the SQLDA.

DB2 does not change this value. For techniques to determine the number of required occurrences, see Allocating the SQLDA.

For REXX: The SQLDA is not allocated before it is used. An SQLDA consists of a set of stem variables. There is one occurrence of variable *stem*.SQLD, followed by zero or more occurrences of a set of variables that is equivalent to an SQLVAR structure. Those variables begin with *stem.n*.

After the DESCRIBE statement is executed, all the fields in the SQLDA except SQLN are either set by DB2 or ignored. For information on the contents of the fields, see The SQLDA contents returned after DESCRIBE.

USING
Indicates what value to assign to each SQLNAME variable in the SQLDA. If the requested value does not exist, SQLNAME is set to a length of 0.

NAMES
Assigns the name of the column. This is the default.

LABELS
Assigns the label of the column. (Column labels are defined by the LABEL statement.)

ANY
Assigns the column label, and if the column has no label, the column name.

BOTH
Assigns both the label and name of the column. In this case, two or three occurrences of SQLVAR per column, depending on whether the result set contains distinct types, are needed to accommodate the additional information. To specify this expansion of the SQLVAR array, set SQLN to 2×*n* or 3×*n*, where *n* is the number of columns in the object being described. For each of the columns, the first *n* occurrences of SQLVAR, which are the base SQLVAR entries, contain the column names. Either the second or third *n* occurrences of SQLVAR, which are the extended SQLVAR entries, contain the column labels. If there are no distinct types, the labels are returned in the second set of SQLVAR entries. Otherwise, the labels are returned in the third set of SQLVAR entries.

Notes

Using PREPARE INTO clause: Information about a prepared statement can also be obtained by using the INTO clause of the PREPARE statement.

Allocating the SQLDA: Before the DESCRIBE or PREPARE INTO statement is executed, the value of SQLN must be set to a value greater than or equal to zero to indicate how many occurrences of SQLVAR are provided in the SQLDA. Also, enough storage must be allocated to contain the number of occurrences that SQLN specifies. To obtain the description of the columns of the result table of a prepared SELECT statement, the number of occurrences of SQLVAR must be at least equal to the number of columns. Furthermore, if USING BOTH is specified, or if the columns include LOBs or distinct types, the number of occurrences of SQLVAR should be two or three times the number of columns. See "Determining how many SQLVAR occurrences are needed" on page 2084 for more information.

First technique: Allocate an SQLDA with enough occurrences of SQLVAR to accommodate any select list that the application will have to process. At the

extreme, the number of SQLVARs could equal three times the maximum number of columns allowed in a result table. After the SQLDA is allocated, the application can use the SQLDA repeatedly.

This technique uses a large amount of storage that is never deallocated, even when most of this storage is not used for a particular select list.

Second technique: Repeat the following two steps for every processed select list:

1. Execute a DESCRIBE statement with an SQLDA that has no occurrences of SQLVAR; that is, an SQLDA for which SQLN is zero.
2. Allocate a new SQLDA with enough occurrences of SQLVAR. Use the values that are returned in SQLD and SQLCODE to determine the number of SQLVAR entries that are needed. The value of SQLD is the number of columns in the result table, which is either the required number of occurrences of SQLVAR or a fraction of the required number (see "Determining how many SQLVAR occurrences are needed" on page 2084 for details). If the SQLCODE is +236, +237, +238, or +239, the number of SQLVAR entries that is needed is two or three times the value in SQLD, depending on whether USING BOTH was specified. Set SQLN to reflect the number of SQLVAR entries that have been allocated.
3. Execute the DESCRIBE statement again, using the new SQLDA.

This technique allows better storage management than the first technique, but it doubles the number of DESCRIBE statements.

Third technique: Allocate an SQLDA that is large enough to handle most (hopefully, all) select lists but is also reasonably small. If an execution of DESCRIBE fails because SQLDA is too small, allocate a larger SQLDA and execute the DESCRIBE statement again.

For the new larger SQLDA, use the values that are returned in SQLD and SQLCODE from the failing DESCRIBE statement to calculate the number of occurrences of SQLVAR that are needed, as described in technique two. Remember to check for SQLCODEs +236, +237, +238, and +239, which indicate whether *extended* SQLVAR entries are needed because the data includes LOBs or distinct types.

This third technique is a compromise between the first two techniques. Its effectiveness depends on a good choice of size for the original SQLDA.

The SQLDA contents returned on DESCRIBE: After a DESCRIBE statement is executed, the following list describes the contents of the SQLDA fields as they are set by DB2 or ignored. These descriptions do not necessarily apply to the uses of an SQLDA in other SQL statements (EXECUTE, OPEN, FETCH). For more on the other uses, see "SQL descriptor area (SQLDA)" on page 2079.

SQLDAID

DB2 sets the first 6 bytes to 'SQLDA ' (5 letters followed by the space character) and the eighth byte to a space character. The seventh byte is set to indicate the number of SQLVAR entries that are needed to describe each column of the result table as follows:

space The value of space occurs when:

- USING BOTH was not specified and the columns being described do not include LOBs or distinct types. Each column

only needs one SQLVAR entry. If the SQL standard option is yes, DB2 sets SQLCODE to warning code +236. Otherwise, SQLCODE is zero.

- USING BOTH was specified and the columns being described do not include LOBs or distinct types. Each column needs two SQLVAR entries. DB2 sets SQLD to two times the number of columns of the result table. The second set of SQLVARs is used for the labels.

2 Each column needs two SQLVAR entries. Two entries per column are required when:

- USING BOTH was not specified and the columns being described include LOBs or distinct types or both. DB2 sets the second set of SQLVAR entries with information for the LOBs or distinct types being described.
- USING BOTH was specified and the columns include LOBs but not distinct types. DB2 sets the second set of SQLVAR entries with information for the LOBs and labels for the columns being described.

3 Each column needs three SQLVAR entries. Three entries are required only when USING BOTH is specified and the columns being described include distinct types. The presence of LOB data does not matter. It is the distinct types and not the LOBs that cause the need for three SQLVAR entries per column when labels are also requested. DB2 sets the second set of SQLVAR entries with information for the distinct types (and LOBs, if any) and the third set of SQLVAR entries with the labels of the columns being described.

A REXX SQLDA does not contain this field.

SQLDABC
The length of the SQLDA in bytes. DB2 sets the value to SQLN×44+16.

A REXX SQLDA does not contain this field.

SQLD If the prepared statement is a query, DB2 sets the value to the number of columns in the object being described (the value is actually twice the number of columns in the case where USING BOTH was specified and the result table does not include LOBs or distinct types). Otherwise, if the statement is not a query, DB2 sets the value to 0.

SQLVAR
An array of field description information for the column being described. There are two types of SQLVAR entries—the base SQLVAR and the extended SQLVAR.

If the value of SQLD is 0, or is greater than the value of SQLN, no values are assigned to any occurrences of SQLVAR. If the value of SQLN was set so that there are enough SQLVAR occurrences to describe the specified columns (columns with LOBs or distinct types and a request for labels increase the number of SQLVAR entries that are needed), the values are assigned to the first *n* occurrences of SQLVAR so that the first occurrence of SQLVAR contains a description of the first column, the second occurrence of SQLVAR contains a description of the second column, and so on. This first set of SQLVAR entries are referred to as *base SQLVAR* entries. Each column always has a base SQLVAR entry.

If the DESCRIBE statement included the USING BOTH clause, or the columns being described include LOBs or distinct types, additional SQLVAR entries are needed. These additional SQLVAR entries are referred to as the *extended SQLVAR* entries. There can be up to two sets of extended SQLVAR entries for each column.

For REXX, the SQLVAR is a set of stem variables that begin with *stem.n*, instead of a structure. The REXX SQLDA uses only a base SQLVAR. The way in which DB2 assigns values to the SQLVAR variables is the same as for other languages. That is, the *stem.*1 variables describe the first column in the result table, the *stem.*2 variables describe the second column in the result table, and so on. If USING BOTH is specified, the *stem.n*+1 variables also describe the first column in the result table, the *stem.n*+2 variables also describe the second column in the result table, and so on.

The base SQLVAR:

SQLTYPE
A code that indicates the data type of the column and whether the column can contain null values. For the possible values of SQLTYPE, see Table 169 on page 2090.

SQLLEN
A length value depending on the data type of the result columns. SQLLEN is 0 for LOB and XML data types. For the other possible values of SQLLEN, see Table 169 on page 2090.

In a REXX SQLDA, for DECIMAL or NUMERIC columns, DB2 sets the SQLPRECISION and SQLSCALE fields instead of the SQLLEN field.

SQLDATA
The CCSID of a string column. For possible values, see Table 170 on page 2093.

In a REXX SQLDA, DB2 sets the SQLCCSID field instead of the SQLDATA field.

SQLIND
Reserved.

SQLNAME
The unqualified name or label of the column, depending on the value of USING (NAMES, LABELS, ANY, or BOTH). The field is a string of length 0 if the column does not have a name or label. For more details on unnamed columns, see the discussion of the names of result columns under "select-clause" on page 765. This value is returned in the encoding scheme specified by the ENCODING bind option for the plan or package that contains the statement.

The extended SQLVAR:

SQLLONGLEN
The length attribute of a BLOB, CLOB, or DBCLOB column.

* Reserved.

SQLDATALEN
Not Used.

SQLDATATYPE-NAME

For a distinct type, the fully qualified distinct type name. Otherwise, the value is the fully qualified name of the built-in data type.

For a label, the label for the column.

This value is returned in the encoding scheme specified by the ENCODING bind option for the plan or package that contains this statement.

The REXX SQLDA does not use the extended SQLVAR.

Performance considerations: Although DB2 does not change the value of SQLN, you might want to reset this value after the DESCRIBE statement is executed. If the contents of SQLDA from the DESCRIBE statement is used in a later FETCH statement, set SQLN to *n* (where *n* is the number of columns of the result table) before executing the FETCH statement. For details, see Preparing the SQLDA for data retrieval.

Preparing the SQLDA for data retrievals: This note is relevant if you are applying DESCRIBE to a prepared query and you intend to use the SQLDA in the FETCH statements you employ to retrieve the result table rows. To prepare the SQLDA for that task, you must set the SQLDATA field of SQLVAR. SQLIND must be set if SQLTYPE is odd, and SQLNAME must be set when overriding the CCSID. For the meaning of those fields in that context, see "SQL descriptor area (SQLDA)" on page 2079.

Also, SQLN and SQLDABC should be reset (if necessary) to *n* and *n*×44+16, where *n* is the number of columns in the result table. Doing so can improve performance when the rows of the result table are fetched.

Support for extended dynamic SQL in a distributed environment: In a distributed environment where DB2 for z/OS is the server and the requester supports extended dynamic SQL, such as DB2 Server for VSE & VM, a DESCRIBE statement that is executed against an SQL statement in the extended dynamic package appears to DB2 as a DESCRIBE statement against a static SQL statement in the DB2 package. A DESCRIBE statement cannot normally be issued against a static SQL statement. However, a DESCRIBE against a static SQL statement that is generated by extended dynamic SQL executes without error if the package has been rebound after field DESCRIBE FOR STATIC on installation panel DSNTIP4 has been set to YES.

YES indicates that DB2 generates an SQLDA for the DESCRIBE at bind time so that DESCRIBE requests for static SQL statements can be satisfied at execution time. For more information, see *DB2 Installation Guide*.

Avoiding double preparation when using REOPT(ALWAYS) or REOPT(ONCE): If bind option REOPT(ALWAYS) or REOPT(ONCE) is in effect, DESCRIBE causes the statement to be prepared if it is not already prepared. If issued before an OPEN or an EXECUTE, the DESCRIBE causes the statement to be prepared without input variables. If the statement has input variables, the statement must be prepared again when it is opened or executed. When REOPT(ONCE) is in effect, the statement is always prepared twice even if there are no input variables. Therefore, to avoid preparing statements twice, issue the DESCRIBE after the OPEN. For non-cursor statements, open and fetch processing are performed on the EXECUTE. So, if a DESCRIBE must be issued, the statement will be prepared twice.

The use of a prepared statement for an EXPLAIN statement can cause duplicate entries in the explain tables when the prepared statement specifies the REOPT(ALWAYS) bind option and is executed using the jcc driver.

Errors occurring on DESCRIBE: In local and remote processing, the DEFER(PREPARE) and REOPT(ALWAYS)/REOPT(ONCE) bind options can cause some errors that are normally issued during PREPARE processing to be issued on DESCRIBE.

Considerations for implicitly hidden columns: A DESCRIBED OUTPUT statement only returns information about implicitly hidden columns if the column (of a base table that is defined as implicitly hidden) is explicitly specified as part of the SELECT list of the final result table of the query described. If implicitly hidden columns are not part of the result table of a query, a DESCRIBE OUTPUT statement that returns information about that query will not contain information about any implicitly hidden columns.

Using host variables: If the DESCRIBE statement contains host variables, the contents of the host variables are assumed to be in the encoding scheme that was specified in the ENCODING parameter when the package or plan that contains the statement was bound.

Considerations for array elements: CCSID UNICODE is returned for a result column that corresponds to a reference to an array element with a datetime data type.

Example

In a PL/I program, execute a DESCRIBE statement with an SQLDA that has no occurrences of SQLVAR. If SQLD is greater than zero, use the value to allocate an SQLDA with the necessary number of occurrences of SQLVAR and then execute a DESCRIBE statement using that SQLDA. This is the second technique described in Allocating the SQLDA.

```
EXEC SQL  BEGIN DECLARE SECTION;
  DCL  STMT1_STR   CHAR(200)  VARYING;
EXEC SQL  END DECLARE SECTION;
EXEC SQL  INCLUDE SQLDA;
EXEC SQL  DECLARE DYN_CURSOR CURSOR FOR STMT1_NAME;
... /* code to prompt user for a query, then to generate */
    /* a select-statement in the STMT1_STR            */
EXEC SQL  PREPARE STMT1_NAME FROM :STMT1_STR;
... /* code to set SQLN to zero and to allocate the SQLDA */
EXEC SQL  DESCRIBE STMT1_NAME INTO :SQLDA;
... /* code to check that SQLD is greater than zero, to set */
    /* SQLN to SQLD, then to re-allocate the SQLDA          */
EXEC SQL  DESCRIBE STMT1_NAME INTO :SQLDA;
... /* code to prepare for the use of the SQLDA             */
EXEC SQL  OPEN DYN_CURSOR;
... /* loop to fetch rows from result table                 */
EXEC SQL  FETCH DYN_CURSOR USING DESCRIPTOR :SQLDA;
.
.
.
```

DESCRIBE PROCEDURE

The DESCRIBE PROCEDURE statement gets information about the result sets returned by a stored procedure. The information, such as the number of result sets, is put into a descriptor.

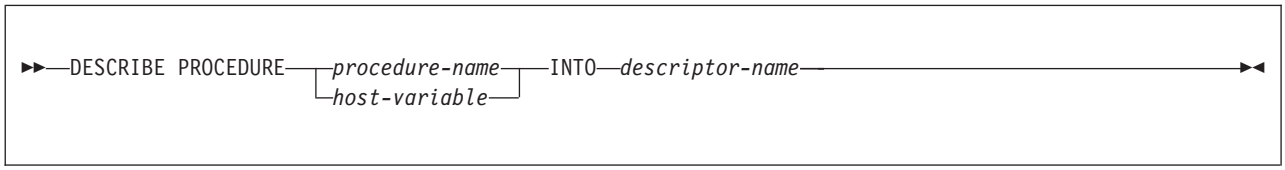
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

None required.

Syntax



Description

procedure-name **or** *host-variable*

Identifies the stored procedure that returned one or more result sets. When the DESCRIBE PROCEDURE statement is executed, the procedure name must identify a stored procedure that the requester has already invoked using the SQL CALL statement. The procedure name can be specified as a one, two, or three-part name. The procedure name in the DESCRIBE PROCEDURE statement must be specified the same way that it was specified on the CALL statement. For example, if a two-part procedure name was specified on the CALL statement, you must specify a two-part procedure name in the DESCRIBE PROCEDURE statement.

If a host variable is used:

- It must be a character string variable with a length attribute that is not greater than 254.
- It must not be followed by an indicator variable.
- The value of the host variable is a specification that depends on the database server. Regardless of the server, the specification must:
  - Be left justified within the host variable
  - Not contain embedded blanks
  - Be padded on the right with blanks if its length is less than that of the host variable

INTO *descriptor-name*

Identifies an SQL descriptor area (SQLDA). The information returned in the SQLDA describes the result sets returned by the stored procedure.

Considerations for allocating and initializing the SQLDA are similar to those for DESCRIBE TABLE.

The contents of the SQLDA after executing a DESCRIBE PROCEDURE statement are:

- The first 5 bytes of the SQLDAID field are set to 'SQLPR'.

  A REXX SQLDA does not contain SQLDAID.
- Bytes 6 to 8 of the SQLDAID field are reserved.
- The SQLD field is set to the total number of result sets. A value of 0 in the field indicates there are no result sets.
- There is one SQLVAR entry for each result set.
- The SQLDATA field of each SQLVAR entry is set to the result set locator value associated with the result set.

  For a REXX SQLDA, SQLLOCATOR is set to the result set locator value.
- The SQLIND field of each SQLVAR entry is set to the estimated number of rows in the result set

  For a REXX SQLDA, the SQLIND field is not used for DESCRIBE.
- The SQLNAME field is set to the name of the cursor used by the stored procedure to return the result set. This value is returned in the encoding scheme specified by the ENCODING bind option for the plan or package that contains this statement.

Notes

SQLDA information: A value of -1 in the SQLIND field indicates that an estimated number of rows in the result set is not provided. DB2 for z/OS always sets SQLIND to -1. For a REXX SQLDA, the SQLIND field is not used for DESCRIBE.

DESCRIBE PROCEDURE does not return information about the parameters expected by the stored procedure.

Assignment of locator values: Locator values are assigned to the SQLVAR entries in the SQLDA in the order that the associated cursors are opened at run time. Locator values are not provided for cursors that are closed when control is returned to the invoking application. If a cursor was closed and later re-opened before returning to the invoking application, the most recently executed OPEN CURSOR statement for the cursor is used to determine the order in which the locator values are returned for the procedure result sets. For example, assume procedure P1 opens three cursors A, B, C, closes cursor B and then issues another OPEN CURSOR statement for cursor B before returning to the invoking application. The locator values are assigned in the order A, C, B.

Alternatively, an ASSOCIATE LOCATORS statement can be used to copy the locator values to result set locator variables.

Using host variables: If the DESCRIBE PROCEDURE statement contains host variables, the contents of the host variables are assumed to be in the encoding scheme that was specified in the ENCODING parameter when the package or plan that contains the statement was bound.

Examples

The statements in the following examples are assumed to be in PL/I programs.

Example 1: Place information about the result sets returned by stored procedure P1 into the descriptor named by SQLDA1. Assume that the stored procedure is called with a one-part name from current server SITE2.

```
EXEC SQL CONNECT TO SITE2;
EXEC SQL CALL P1;
EXEC SQL DESCRIBE PROCEDURE P1 INTO :SQLDA1;
```

Example 2: Repeat the scenario in Example 1, but use a two-part name to specify an explicit schema name for the stored procedure to ensure that stored procedure P1 in schema MYSCHEMA is used.

```
EXEC SQL CONNECT TO SITE2;
EXEC SQL CALL MYSCHEMA.P1;
EXEC SQL DESCRIBE PROCEDURE MYSCHEMA.P1 INTO :SQLDA1;
```

Example 3: Place information about the result sets returned by the stored procedure identified by host variable HV1 into the descriptor named by SQLDA2. Assume that host variable HV1 contains the value SITE2.MYSCHEMA.P1 and the stored procedure is called with a three-part name.

```
EXEC SQL CALL SITE2.MYSCHEMA.P1;
EXEC SQL DESCRIBE PROCEDURE :HV1 INTO :SQLDA2;
```

The preceding example would be invalid if host variable HV1 had contained the value MYSCHEMA.P1, a two-part name. For the example to be valid with that two-part name in host variable HV1, the current server must be the same as the location name that is specified on the CALL statement as the following statements demonstrate. This is the only condition under which the names do not have to be specified the same way and a three-part name on the CALL statement can be used with a two-part name on the DESCRIBE PROCEDURES statement.

```
EXEC SQL CONNECT TO SITE2;
EXEC SQL CALL SITE2.MYSCHEMA.P1;
EXEC SQL ASSOCIATE LOCATORS (:LOC1, :LOC2)
         WITH PROCEDURE :HV1;
```

DESCRIBE TABLE

The DESCRIBE TABLE statement obtains information about a designated table or view.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

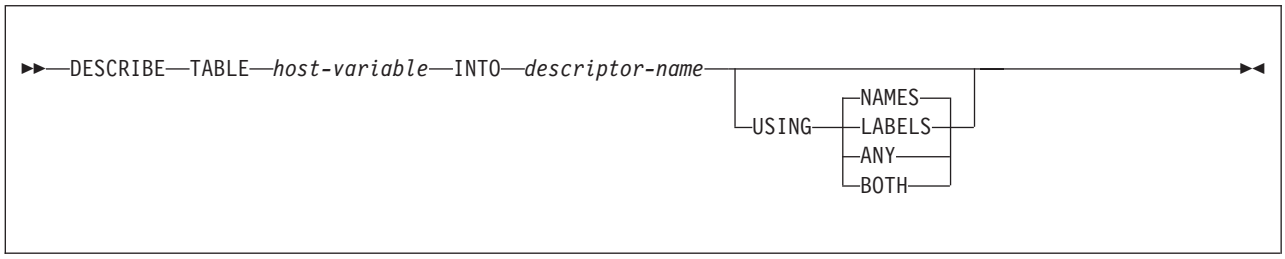
The privileges that are held by the authorization ID that owns the plan or package must include at least one of the following (if there is a plan, authorization checking is done only against the plan owner):

- Ownership of the table or view
- The SELECT, INSERT, UPDATE, DELETE, or REFERENCES privilege on the object
- The ALTER or INDEX privilege on the object (tables only)
- DBADM authority over the database that contains the object (tables only)
- SYSADM authority
- SYSCTRL authority (catalog tables only)
- ACCESSCTRL authority (catalog tables only)
- System DBADM
- DATAACCESS authority
- EXPLAIN authority
- SQLADM authority

If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

For an RRSAF application that does not have a plan and in which the requester and the server are DB2 for z/OS systems, authorization to execute the package is performed against the primary or secondary authorization ID of the process.

Syntax



Description

TABLE *host-variable*
Identifies the table or view. The name must not identify an auxiliary table. When the DESCRIBE statement is executed, the host variable must contain a name which identifies a table or view that exists at the current server. This

variable must be a fixed-length or varying-length character string with a length attribute less than 256. The name must be followed by one or more blanks if the length of the name is less than the length of the variable. It cannot contain a period as the first character and it cannot contain embedded blanks. In addition, the quotation mark is the escape character regardless of the value of the string delimiter option. An indicator variable must not be specified for the host variable.

INTO *descriptor-name*
Identifies an SQL descriptor area (SQLDA), which is described in "SQL descriptor area (SQLDA)" on page 2079. See "Identifying an SQLDA in C or C++" on page 2099 for how to represent *descriptor-name* in C.

For languages other than REXX: Before the DESCRIBE statement is executed, the user must set the following variable in the SQLDA and the SQLDA must be allocated.

SQLN Indicates the number of SQLVAR occurrences provided in the SQLDA. DB2 does not change this value. For techniques to determine the number of required occurrences, see Allocating the SQLDA.

For REXX: The SQLDA is not allocated before it is used. An SQLDA consists of a set of stem variables. There is one occurrence of variable *stem.*SQLD, followed by zero or more occurrences of a set of variables that is equivalent to an SQLVAR structure. Those variables begin with *stem.n*.

After the DESCRIBE statement is executed, all the fields in the SQLDA except SQLN are either set by DB2 or ignored. For information on the contents of the fields, see The SQLDA contents returned after DESCRIBE.

USING
Indicates what value to assign to each SQLNAME variable in the SQLDA. If the requested value does not exist, SQLNAME is set to a length of 0.

NAMES
Assigns the name of the column. This is the default.

LABELS
Assigns the label of the column. (Column labels are defined by the LABEL statement.)

ANY
Assigns the column label, and if the column has no label, the column name.

BOTH
Assigns both the label and name of the column. In this case, two or three occurrences of SQLVAR per column, depending on whether the result set contains distinct types, are needed to accommodate the additional information. To specify this expansion of the SQLVAR array, set SQLN to 2×*n* or 3×*n*, where *n* is the number of columns in the object being described. For each of the columns, the first *n* occurrences of SQLVAR, which are the base SQLVAR entries, contain the column names. Either the second or third *n* occurrences of SQLVAR, which are the extended SQLVAR entries, contain the column labels. If there are no distinct types, the labels are returned in the second set of SQLVAR entries. Otherwise, the labels are returned in the third set of SQLVAR entries.

For a declared temporary table, the name of the column is assigned regardless of the value specified in the USING clause because declared temporary tables cannot have labels.

Notes

See "DESCRIBE OUTPUT" on page 1596 for information about the following topics:

- Allocating the SQLDA
- The SQLDA contents that are returned on DESCRIBE
- Performance considerations for DESCRIBE
- Using host variables in DESCRIBE statements

Considerations for implicitly hidden columns: A DESCRIBE TABLE statement does return information about implicitly hidden columns in tables.

DROP

The DROP statement removes an object at the current server. Except for storage groups, any objects that are directly or indirectly dependent on that object are deleted. Whenever an object is deleted, its description is deleted from the catalog at the current server, and any packages that refer to the object are invalidated.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

To drop the following objects, the privilege set must include at least one of the listed authorities or privileges:

Table, table space, or index:

- Ownership of the object (for an index, the owner is the owner of the table or index)
- DBADM authority
- SYSADM or SYSCTRL authority
- System DBADM

If the table space is in a database that is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

Database:

- The DROP privilege on the database
- DBADM or DBCTRL authority for the database
- SYSADM or SYSCTRL authority
- System DBADM

If the database is implicitly created, the privileges must be on the implicit database or on DSNDB04.

Storage group, or view:

- Ownership of the object
- SYSADM or SYSCTRL authority
- System DBADM

Alias for a table or view:

- Ownership of the object
- SYSADM or SYSCTRL authority
- System DBADM

Alias for a sequence:

- Ownership of the object
- The DROPIN privilege on the schema
- SYSADM or SYSCTRL authority
- System DBADM

Package:

- Ownership of the package

- The BINDAGENT privilege granted from the package owner
- PACKADM authority for the collection or for all collections
- SYSADM or SYSCTRL authority

Synonym:
Ownership of the synonym

Role or trusted context:
- Ownership of the object
- SYSADM or SYSCTRL authority
- SECADM

If the installation parameter SEPARATE SECURITY is NO, SYSADM authority has implicit SECADM and SYSCTRL authority and can drop a role or trusted context.

Row permission or column mask:
At least SECADM authority

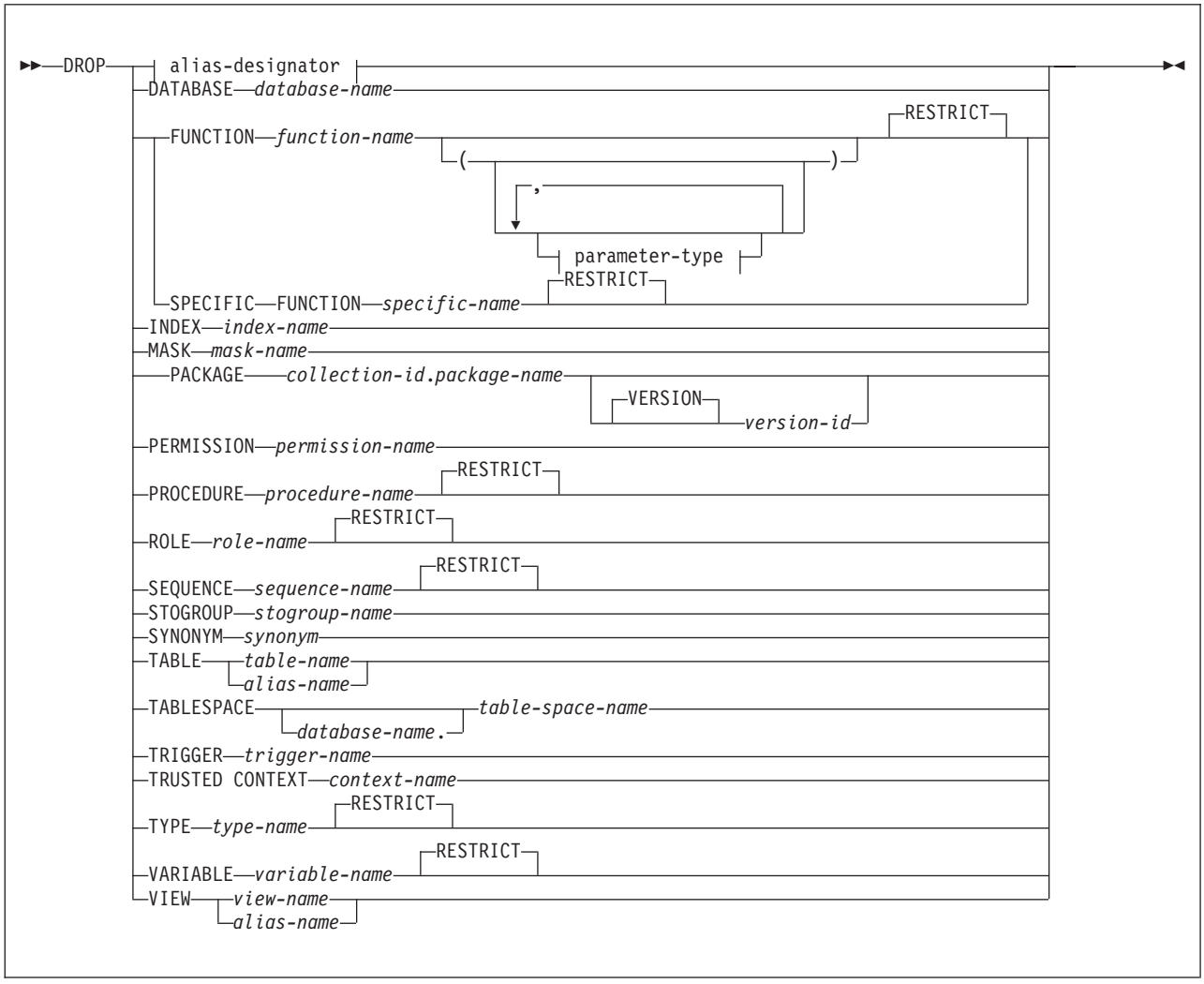
Distinct type, stored procedure, trigger, user-defined function, global variable, or sequence:
- Ownership of the object [34]
- The DROPIN privilege on the schema
- SYSADM or SYSCTRL authority
- System DBADM

The authorization ID that matches the schema name implicitly has the DROPIN privilege on the schema.

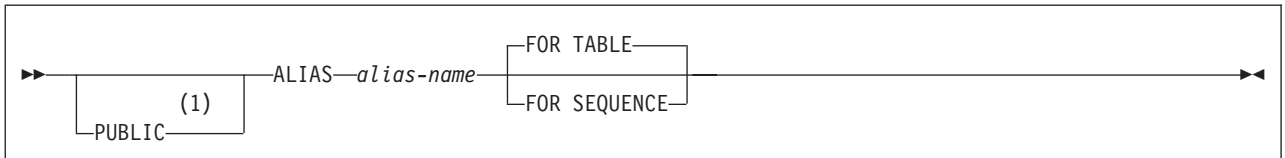
Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set is the union of the privilege sets that are held by each authorization ID of the process. If running in a trusted context with a role, the privilege set also includes those privileges that are held by the role that is associated with the primary authorization ID. However, the implicit schema match does not apply to the role when determining if DROPIN schema privilege is held.

34. Not applicable for stored procedures defined in releases of DB2 for z/OS prior to Version 6.

Syntax



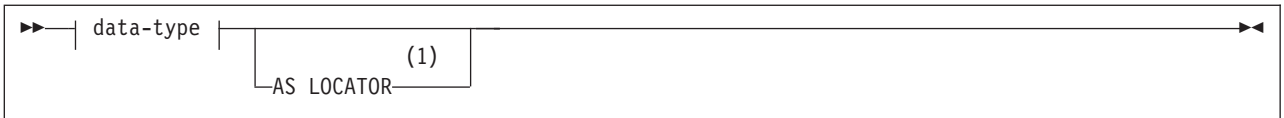
alias-designator:



Notes:

1 If PUBLIC is specified, FOR SEQUENCE must also be specified.

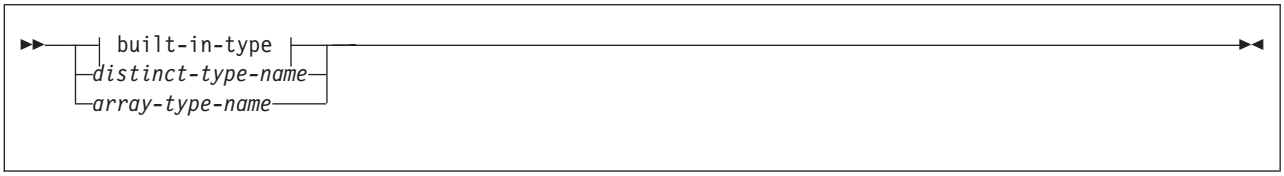
parameter type:



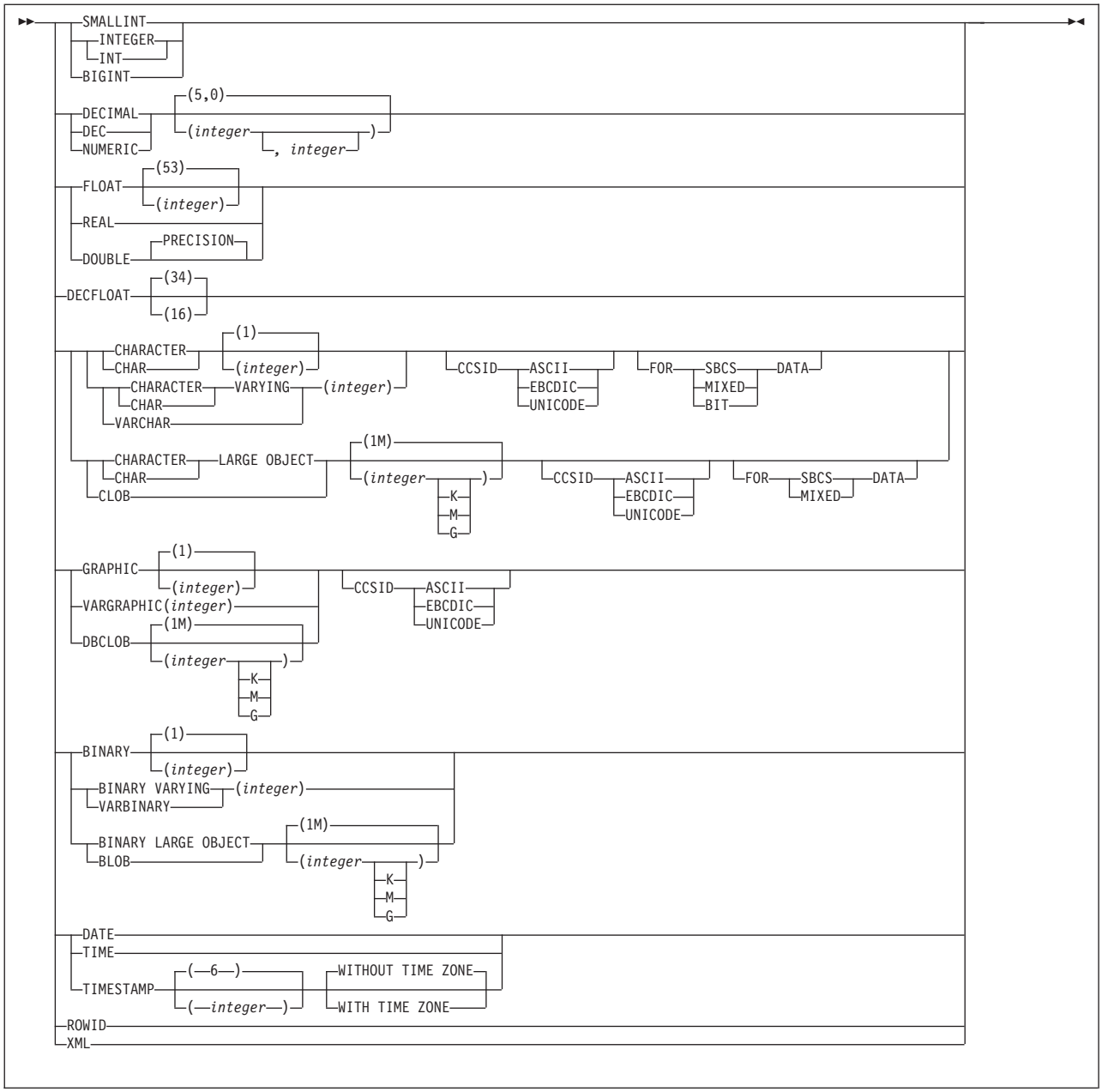
Notes:

1 AS LOCATOR can be specified only for a LOB data type or a distinct type based on a LOB data type.

data type:



built-in-type:



Description

alias-designator

PUBLIC
Specifies that the alias to be dropped is a public alias. The *alias-name* must identify an alias that exists in the SYSPUBLIC schema.

If the PUBLIC keyword is specified, *alias-name* must identify a public alias that exists at the current server.

ALIAS *alias-name*
Identifies the alias to be dropped. The *alias-name* must identify an alias that exists at the current server.

Dropping an alias for a table or view has no effect on any view, materialized query table, or synonym that was defined using the alias. If the alias is referenced in the definition of a row permission or a column mask, it cannot be dropped.

Dropping an alias for a sequence has no effect on any view or materialized query table that was defined using the alias. If the alias is referenced in the definition of an inline SQL function, it cannot be dropped. When an alias for a sequence is dropped, all packages that refer to the sequence alias are invalidated.

If the alias is referenced in the definition of a row permission or a column mask, the alias cannot be dropped.

FOR TABLE
Specifies that the alias to be dropped is for a table or view. Dropping an alias for a table has no effect on any view, materialized query table, or synonym that was defined using the alias.

FOR SEQUENCE
Specifies that the alias to be dropped is for a sequence. Dropping an alias for a sequence has no effect on any view, or materialized query table that was defined using the alias.

DATABASE *database-name*
Identifies the database to drop. The name must identify a database that exists at the current server. DSNDB04 or DSNDB06 must not be specified. The privilege set must include SYSADM authority.

Whenever a database is dropped, all of its table spaces, tables, index spaces, and indexes are also dropped. Any pending changes to the definitions of the table spaces and indexes in the database are also dropped.

You can drop a database that contains a history table only if the database also contains the associated system-period temporal table. You can drop a database that contains a system-period temporal table when the associated history table is in another database. In this case, the action cascades to drop the history table in the other database.

You can drop a database that contains an archive table only if the database also contains the associated archive-enabled table. You can drop a database that contains an archive-enabled table when the associated archive table is contained in another database. In this case, the action cascades to drop the archive table in the other database.

FUNCTION or SPECIFIC FUNCTION
Identifies the function to drop. The function must exist at the current server, and it must have been defined with the CREATE FUNCTION statement. The particular function can be identified by its name, function signature, or specific name.

Functions that are implicitly generated by the CREATE TYPE statement cannot be dropped using the DROP statement. They are implicitly dropped when the distinct type is dropped.

As indicated by the default keyword RESTRICT, the function is not dropped if any of the following dependencies exist:

- Another function is sourced on the function.
- A view uses the function.
- A trigger package uses the function.

- The definition of a materialized query table uses the function.
- The definition of a row permission or a column mask uses the function.

When a function is dropped, all privileges on the function are also dropped. Any packages that are dependent on the function dropped are made inoperative.

FUNCTION *function-name*
Identifies the function by its name. The *function-name* must identify exactly one function. The function can have any number of parameters defined for it. If there is more than one function of the specified name in the specified or implicit schema, an error is returned.

FUNCTION *function-name (parameter-type,...)*
Identifies the function by its function signature, which uniquely identifies the function. The *function-name (parameter-type, ...)* must identify a function with the specified function signature. The specified parameters must match the data types in the corresponding position that were specified when the function was created. The number of data types, and the logical concatenation of the data types is used to identify the specific function instance which is to be dropped. Synonyms for data types are considered a match.

If the function was defined with a table parameter (the LIKE TABLE *name* AS LOCATOR clause was specified in the CREATE FUNCTION statement to indicate that one of the input parameters is a transition table), the function signature cannot be used to uniquely identify the function. Instead, use one of the other syntax variations to identify the function with its function name, if unique, or its specific name.

If *function-name ()* is specified, the function identified must have zero parameters.

function-name
Identifies the name of the function.

(parameter-type,...)
Identifies the parameters of the function.

If an unqualified distinct type name is specified, DB2 searches the SQL path to resolve the schema name for the distinct type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parentheses indicate that the database manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a function defined with a data type of DEC(7,2). Similarly DECFLOAT() will be considered a match for DECFLOAT(16) or DECFLOAT(34). However, FLOAT cannot be specified with empty parenthesis because its parameter value indicates a specific data type (REAL or DOUBLE).
- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).

- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

For data types with a subtype or encoding scheme attribute, specifying the FOR *subtype* DATA clause or the CCSID clause is optional. Omission of either clause indicates that DB2 ignores the attribute when determining whether the data types match. If you specify either clause, it must match the value that was implicitly or explicitly specified in the CREATE FUNCTION statement.

AS LOCATOR
Specifies that the function is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or a distinct type based on a LOB.

SPECIFIC FUNCTION *specific-name*
Identifies the function by its specific name. The *specific-name* must identify a specific function that exists at the current server.

INDEX *index-name*
Identifies the index to drop. The name must identify a user-defined index that exists at the current server but must not identify a populated index on an auxiliary table or an index that was implicitly created for a table that contains an XML column. (For details on dropping user-defined indexes on catalog tables, see "SQL statements allowed on the catalog" on page 2113.) A populated index on an auxiliary table can only be dropped by dropping the base table. The name must not identify an auxiliary table for an object that is involved in a clone relationship.

If the index that is dropped was created by specifying the ENDING AT clause to define partition boundaries, the table is converted to use table-controlled partitioning. The high limit key for the last partition is set to the highest possible value for ascending key columns or the lowest possible value for descending key columns.

Whenever an index is directly or indirectly dropped, its index space is also dropped. The name of a dropped index space cannot be reused until a commit operation is performed. Any pending changes to the definitions of the index is also dropped.

If the index is a unique index used to enforce a unique constraint (primary or unique key), the unique constraint must be dropped before the index can be dropped. In addition, if a unique constraint supports a referential constraint, the index cannot be dropped unless the referential constraint is dropped.

However, a unique index (for a unique key only) can be dropped without first dropping the unique key constraint if the unique key was created in a release of DB2 before Version 7 and if the unique key constraint has no associated referential constraints. For information about dropping constraints, see "ALTER TABLE" on page 984.

If the table space is explicitly created and a unique index is dropped and that index was defined on a ROWID column that is defined as GENERATED BY DEFAULT, the table can still be used, but rows cannot be inserted into that table.

If the table space is implicitly created, the index cannot be dropped if it is defined on a ROWID column that is defined as GENERATED BY DEFAULT.

If an empty index on an auxiliary table is dropped, the base table is marked incomplete. If the base table space is implicitly created, the index on an auxiliary table cannot be dropped.

Drop index will result in the deletion of rows in the SYSCOLDIST and SYSCOLDISTATS catalog tables if no other indexes on the table have the same column group in their key sequence prefix.

MASK *mask-name*
Identifies the column mask to drop. The name must identify a column mask that exists at the current server.

PACKAGE *collection-id.package-name*
Identifies the package version to drop. The name plus the implicitly or explicitly specified *version-id* must identify a package version that exists at the current server. Omission of the *version-id* is an implicit specification of the null version.

The name must not identify a trigger package or a package that is associated with an SQL routine. A trigger package can only be dropped by dropping the associated trigger or subject table. A package that is associated with a native SQL procedure can only be dropped with an ALTER PROCEDURE statement with a DROP VERSION clause that specifies the particular version that is to be dropped, or with a DROP PROCEDURE statement if it is the only version that is defined for the procedure.

Specify this clause to drop a package that is created as the result of a BIND COPY command used to deploy a version of a native SQL procedure.

If a package has current, previous, and original copies, the DROP statement will drop all copies.

VERSION *version-id*
version-id is the version identifier that was assigned to the package's DBRM when the DBRM was created. If *version-id* is not specified, a null version is used as the version identifier.

Delimit the version identifier when it:

- Is generated by the VERSION(AUTO) precompiler option
- Begins with a digit
- Contains lowercase or mixed-case letters

For more on version identifiers, see the information on preparing an application program for execution in *DB2 Application Programming and SQL Guide*.

PERMISSION *permission-name*
Identifies the row permission to drop. The name must identify a row permission that exists at the current server. The name must not identify the default row permission that was created implicitly by DB2.

PROCEDURE *procedure-name*
Identifies the stored procedure to drop. The name must identify a stored procedure that has been defined with the CREATE PROCEDURE statement at the current server.

All versions of the native SQL procedure are dropped; all privileges on the procedure are also dropped. In addition, any packages that are dependent on the procedure are marked invalid.

If the procedure is a native SQL procedure, use an ALTER PROCEDURE statement with the DROP VERSION clause to drop a specific version of a

procedure. Use a DROP PACKAGE statement to drop a package for a version of the procedure that is created using the BIND COPY command.

ROLE *role-name*

Identifies the role to drop. *role-name* must identify a role that exists at the current server.

When a role is dropped, all privileges and authorities that have been previously granted to that role are revoked. If the role that is dropped is the owner of statements in the dynamic statement cache, the cached statements are invalidated.

The role is not dropped if any REVOKE restrictions are encountered. REVOKE restrictions include the following:

- Restrictions that are encountered when dependent privileges are included when the privileges of a role are revoked.
- The role is the grantor of any privilege or authority that used ACCESSCTRL or SECADM authority to perform the grant.

If RESTRICT is specified, the role is not dropped is any of the following dependencies exist:

- The role is associated with any trusted context or any user in a trusted context.
- The role is associated with a currently running thread.
- The role is the owner of any of the following objects:

Alias	Row permission
Array type	Sequence
Column mask	Storage group
Database	Stored procedure
Distinct type	Table
Index	Table space
JAR file	Trigger
Materialized query table	Trusted context
Package	User-defined function
Role	View

SEQUENCE *sequence-name*

Identifies the sequence to drop. The name must identify an existing sequence at the current server.

sequence-name must not be the name of an internal sequence object that is used by DB2 (including an implicitly generated sequence for a DB2_GENERATED_DOCID_FOR_XML column). Sequences that are generated by the system for identity columns or implicitly created databases cannot be dropped by using the DROP SEQUENCE statement. A sequence object for an identity column is implicitly dropped when the table that contains the identify column is dropped.

The default keyword RESTRICT indicates that the sequence is not dropped if any of the following dependencies exist:

- A trigger that uses the sequence in a NEXT VALUE or PREVIOUS VALUE expression exists.
- An inline SQL function that uses the sequences in a NEXT VALUE or PREVIOUS VALUE expression exists.

Whenever a sequence is dropped, all privileges on the sequence are also dropped, and the packages that refer to the sequence are invalidated. Dropping a sequence, even if the drop process is rolled back, results in the loss of the still-unassigned cache values for the sequence.

STOGROUP *stogroup-name*
Identifies the storage group to drop. The name must identify a storage group that exists at the current server but not a storage group that is used by any table space or index space.

For information on the effect of dropping the default storage group of a database, see Dropping a default storage group.

SYNONYM *synonym*
Identifies the synonym to drop. In a static DROP SYNONYM statement, the name must identify a synonym that is owned by the owner of the plan or package. In a dynamic DROP SYNONYM statement, the name must identify a synonym that is owned by the SQL authorization ID. Thus, using interactive SQL, a user with SYSADM authority can drop any synonym by first setting CURRENT SQLID to the owner of the synonym.

Dropping a synonym has no effect on any view, materialized query table, or alias that was defined using the synonym, nor does it invalidate any packages that use such views, materialized query tables, or aliases.

If the synonym is referenced in the definition of a row permission or a column mask, it cannot be dropped.

TABLE *table-name* **or** *alias-name*
Identifies the table to drop. The name must identify a table that exists at the current server. It must not identify a catalog table, a table in a partitioned table space, a table that is implicitly created for an XML column, or a populated auxiliary table. A table in a partitioned table space can be dropped only by dropping the table space. A populated auxiliary table or a table that is implicitly created for an XML column can be dropped only by dropping the associated base table.

If *alias-name* is specified, the actual table is dropped as if *table-name* were specified. However, the alias is not dropped. It can be dropped by using the DROP ALIAS statement.

When a table is directly or indirectly dropped, the following items are also dropped:

- All privileges on the table
- All referential constraints in which the table is a parent or dependent
- All synonyms, views, and indexes that are defined on the table
- All row permissions (including the default row permission)
- All column masks that are created for the table

If the table space for the table was implicitly created, it is also dropped. However, if the containing database was implicitly created, it is not dropped. Any pending changes to the definitions of the dropped table space and indexes are also dropped.

For more information, see Dropping an implicitly created database.

When a table is directly or indirectly dropped, all materialized query tables that are defined on the table are also dropped. When a materialized query table is directly or indirectly dropped, the following items are also dropped:

- All privileges on the materialized query table

- All synonyms, views, and indexes that are defined on the materialized query table

Any alias that is defined on the materialized query table is not dropped. Any packages that are dependent on the dropped materialized query table are marked invalid.

You cannot use DROP TABLE to drop a clone table. You must use the ALTER TABLE statement with the DROP CLONE clause to drop a clone table. If a base table that is involved in a clone relationship is dropped, the associated clone table is also dropped. You cannot drop an auxiliary table for an object that is involved in a clone relationship.

The table cannot be dropped if it is defined as a history table for a system-period temporal table.

The table cannot be dropped if it is referenced in the definition of a row permission or a column mask.

To drop a system-period temporal table, the privilege set must also contain the authorization that is required to drop the history table. The history table is dropped when a system-period temporal table is dropped.

If a table with LOB columns is dropped, the auxiliary tables that are associated with the table and the indexes on the auxiliary tables are also dropped. Any LOB table spaces that were implicitly created for the auxiliary tables are also dropped.

If a table with XML columns is dropped, all implicitly created objects for all XML columns are also dropped.

If an empty auxiliary table is dropped, the definition of the base table is marked incomplete. If the base table space is implicitly created, the auxiliary table cannot be dropped.

If the table has a security label column, the primary authorization ID of the DROP statement must have a valid security label, and the RACF SECLABEL class must be active.

If a table that uses hash organization is dropped, all catalog entries for the hash organization are cleaned up.

If an archive-enabled table is dropped, the archive table and any indexes that are defined on the archive table are also dropped. To drop an archive-enabled table, the privilege set must also contain the authorization that is required to drop the archive table. An archive table cannot be explicitly dropped by using the DROP statement.

TABLESPACE *database-name.table-space-name*
Identifies the table space to drop. The name must identify a table space that exists at the current server. The database name must not be DSNDB06. Omission of the database name is an implicit specification of DSNDB04. *table-space-name* must not identify a table space that is implicitly created for an XML column.

Whenever a table space is directly or indirectly dropped, all the tables in the table space are also dropped. The name of a dropped table space cannot be reused until a commit operation is performed. Any pending changes to the definitions of the table space and its indexes are also dropped.

A LOB table space can be dropped only if it does not contain an auxiliary table. If the LOB table space is implicitly created, it cannot be dropped.

Whenever a base table space that contains tables with LOB columns is dropped, all the auxiliary tables and indexes on those auxiliary tables that are associated with the base table space are also dropped.

Whenever a base table space that contains tables with XML columns is dropped, all implicitly created objects for all XML columns are also dropped.

The table space cannot be dropped if it contains a history table or an archive table.

TRIGGER *trigger-name*
Identifies the trigger to drop. The name must identify a trigger that exists at the current server.

Whenever a trigger is directly or indirectly dropped, all privileges on the trigger are also dropped and the associated trigger package is freed. The name of that trigger package is the same as the trigger name and the collection ID is the schema name.

When an INSTEAD OF trigger is dropped, the associated privilege is revoked from anyone that possesses the privilege as a result of an implicit grant that occurred when the trigger is created.

Dropping triggers causes certain packages to be marked invalid. For example, if *trigger-name* specifies an INSTEAD OF trigger on a view V, another trigger might depend on *trigger-name* through an update to the view V, and that trigger package is invalidated.

If a trigger has current, previous, and original copies, the DROP statement will drop all copies.

TRUSTED CONTEXT *context-name*
Identifies the trusted context to drop. The *context-name* must identify a trusted context that exists at the current server. When a trusted context is dropped, all associations to attributes (IP addresses, job names) and associations to users of the trusted context are dropped. If the trusted context is dropped while trusted connections for the context are active, the connections remain active until they terminate or the next attempt at reuse is made.

TYPE *type-name*
Identifies the user-defined type to drop. The name must identify a user-defined type that exists at the current server. The default keyword RESTRICT indicates that the user-defined type is not dropped if any of the following dependencies exist:

- The definition of a column of a table uses the user-defined type.
- The definition of an input or result parameter of a user-defined function uses the user-defined type.
- The definition of a parameter of a stored procedure uses the user-defined type.
- The definition of an extended index uses a cast function that is implicitly generated for the user-defined type.
- The definition of an SQL variable in a procedure or function uses the user-defined type.
- The definition of a row permission or a column mask uses the user-defined type.
- A sequence exists for which the data type of the sequence is the user-defined type.

- One of the following dependencies exists on one of the cast functions that are generated for the user-defined type:
  - Another function is sourced from one of the cast functions
  - A view uses one of the cast functions
  - A trigger package uses one of the cast functions
  - The definition of a materialized query table uses one of the cast functions

Whenever a user-defined type is dropped, all privileges on the distinct type are also dropped. In addition, the cast functions that were generated when the user-defined type was created and the privileges on those cast functions are also dropped.

VARIABLE *variable-name*
Identifies the global variable to drop. The name must identify a global variable that exists at the current server. The name must not identify a built-in global variable. The default keyword RESTRICT indicates that the global variable is not dropped if any of the following dependencies exist:

- The definition of a function, trigger, or view is dependent on the global variable

Packages that are dependent on the global variable are marked invalid when the global variable is dropped. If a statement that is in the dynamic statement cache depends on the global variable and the global variable is dropped, the statement in the dynamic statement cache will be invalidated if it is not in use.

VIEW *view-name* **or** *alias-name*
Identifies the view to drop. The name must identify a view that exists at the current server.

Whenever a view is directly or indirectly dropped, all privileges on the view and all synonyms and views that are defined on the view are also dropped. Whenever a view is directly or indirectly dropped, all materialized query tables defined on the view are also dropped.

If *alias-name* is specified, the actual view will be dropped as if *view-name* were specified. However, the alias is not dropped and can be dropped using the DROP ALIAS statement.

If the view is referenced in the definition of a row permission or a column mask, it cannot be dropped.

Notes

Restrictions on DROP:
DROP is subject to these restrictions:

- DROP DATABASE cannot be performed while a DB2 utility has control of any part of the database.
- DROP INDEX cannot be performed while a DB2 utility has control of the index or its associated table space.
- DROP INDEX cannot be performed if the index is a unique index that is defined on a ROWID column that is defined as GENERATED BY DEFAULT and there are pending changes to the definition of the table space or to any objects within the table space that are explicitly created.
- DROP INDEX cannot be performed if the index is an empty index on an auxiliary table that resides in an explicitly created LOB table space and there are pending changes to the definition of the base table space or to any objects within the base table space.

- DROP INDEX cannot be performed if the index is the hash overflow index for a table that uses hash organization.
- DROP TABLE cannot be performed while a DB2 utility has control of the table space that contains the table.
- DROP TABLE cannot be performed if the table space was explicitly created and there are pending changes to the definition of the table space.
- DROP TABLE cannot be performed if the table is an empty auxiliary table and there are any pending changes to the definition of the base table space or to any objects within the base table space.
- DROP TABLESPACE cannot be performed while a DB2 utility has control of the table space.

In a data sharing environment, the following restrictions also apply:

- If any member has an active resource limit specification table (RLST) you cannot drop the database or table space that contains the table, the table itself, or any index on the table.
- If the member executing the drop cannot access the DB2-managed data sets, only the catalog and directory entries for those data sets are removed.

Objects that have certain dependencies cannot be dropped. For information on these restrictions, see Table 138 on page 1627.

Recreating objects:
After an index or table space is dropped, a commit must be performed before the object can be re-created with the same name. If a table that was created without an IN clause (thereby causing a table space to be implicitly created) is dropped, a table cannot be re-created with the same name until a commit is performed.

Dropping a parent table:
DROP is not DELETE and therefore does not involve delete rules.

Dropping a default storage group:
If you drop the default storage group of a database, the database no longer has a legitimate default. You must then specify USING in any statement that creates a table space or index in the database. You must do this until you either:

- Create another storage group with the same name using the CREATE STOGROUP statement, or
- Designate another default storage group for the database using the ALTER DATABASE statement.

Dropping an implicitly created database:
When a table that resides in an implicitly created table space is dropped, the implicitly created table space and related objects are dropped. However, the implicitly created database is not dropped. This can result in a large number of empty databases in a system. These databases might be eventually reused for newly created implicit table spaces. These implicitly created databases can be dropped using DROP DATABASE.

Dropping a table space or index:
To drop a table space or index, the size of the buffer pool associated with the table space or index must not be zero.

Dropping a LOB table space:
If the base table space is explicitly created, both explicitly created LOB

table spaces and implicitly created LOB table spaces can be dropped if it does not contain any auxiliary tables. If the LOB table space is implicitly created, it will be dropped automatically when the auxiliary table is dropped. If the LOB table space is explicitly created, it is not dropped when the auxiliary table is dropped, and can be explicitly dropped later.

If the base table space is implicitly created, the LOB table space cannot be dropped. If the LOB table space is explicitly created, it can be dropped when the auxiliary table is dropped. The following table shows the relationship between the base table space, the LOB table space, and the use of DROP for the LOB table space and base table space:

Table 136. Use of DROP for LOB table space

How base table was created	How LOB table space was created	Whether DROP can be used on LOB table space	State of LOB table space if base table space is dropped
Explicitly	Explicitly	Yes	LOB table space remains
Explicitly	Implicitly	Yes	LOB table space is dropped
Implicitly	Explicitly	Yes	LOB table space remains
Implicitly	Implicitly	No	N/A

Dropping a database when data sets for DB2 objects have already been deleted:
When some of the data sets for DB2 objects that associated with the database have already been deleted, DROP DATABASE will perform in the following manner:

For DB2-managed objects:
The DROP DATABASE statement will delete the underlying data sets if they exist. If the data sets do not exist, DROP DATABASE will delete only the catalog entries for those data sets.

For user-managed objects:
The DROP DATABASE statement will delete only the catalog entries for the data sets. The underlying data sets will need to be manually deleted after the DROP DATABASE statement is complete.

Dropping a table space in a work file database:
If one member of a data sharing group drops a table space in a work file database, or an entire work file database, that belongs to another member, DB2-managed data sets that the executing member cannot access are not dropped. However, the catalog and directory entries for those data sets are removed.

Dropping resource limit facility (governor) indexes, tables, and table spaces:
While the RLST is active, you cannot issue a DROP DATABASE, DROP INDEX, DROP TABLE, or DROP TABLESPACE statement for an object associated with an RLST that is active on any member of a data sharing group. See Resource limit facility implications for data sharing (DB2 Data Sharing Planning and Administration) for details.

Dropping a temporary table:
To drop a created temporary table or a declared temporary table, use the DROP TABLE statement.

Dropping a materialized query table:
To drop a materialized query table, use the DROP TABLE statement.

Dropping an alias:
Dropping a table or view does not drop its aliases. However, if you use the DROP TABLE statement and specify an alias for a table or view, the table or view will be dropped. To drop an alias, use the DROP ALIAS statement.

Dropping a table from an implicitly created table space:
If you drop a table from an implicitly created table space, the following related objects are also dropped:

- The enforcing primary and unique key indexes
- Any LOB table spaces, auxiliary tables, and auxiliary indexes
- The ROWID index (if the ROWID column is defined as GENERATED BY DEFAULT)

If any LOB columns are defined on the table, the LOB table space is dropped if it was implicitly created. You can use the DROP statement to drop a LOB table space only if one of the following conditions it true:

- The base table space is explicitly created
- The base table space is implicitly created but the LOB table space is explicitly created

You cannot use the DROP statement to drop a LOB table space if both the base table space and the LOB table space are implicitly created.

Dropping an index on an auxiliary table and an auxiliary table:
You can explicitly drop an empty index on an auxiliary table with the DROP INDEX statement, unless the base table space is implicitly created. An empty or populated index on an auxiliary table is implicitly dropped when:

- The auxiliary table is empty and it is explicitly dropped (empty indexes only).
- The associated base table for the auxiliary table is dropped.
- The base table space that contains the associated base table is dropped.

You can explicitly drop an empty auxiliary table with the DROP TABLE statement, unless the base table space is implicitly created. An empty or populated auxiliary table is implicitly dropped when:

- The associated base table for the auxiliary table is dropped.
- The base table space that contains the associated base table is dropped.

The following table shows which DROP statements implicitly or explicitly cause an auxiliary table and the index on that table to be dropped, as indicated by the 'D' in the column.

Table 137. Effect of various DROP statements on auxiliary tables and indexes that are in explicitly created table spaces

	Auxiliary table		Index on auxiliary table	
Statement	**Populated**	**Empty**	**Populated**	**Empty**
DROP TABLESPACE (base table space)	D	D	D	D
DROP TABLE (base table)	D	D	D	D
DROP TABLE (auxiliary table)		D		D

Table 137. Effect of various DROP statements on auxiliary tables and indexes that are in explicitly created table spaces (continued)

	Auxiliary table		Index on auxiliary table	
Statement	**Populated**	**Empty**	**Populated**	**Empty**
DROP INDEX (index on auxiliary table)				D

Note: D indicates that the table or index is dropped.

Dropping a migrated index or table space:
Here, "migration" means migrated by the Hierarchical Storage Manager (DFSMShsm). DB2 does not wait for any recall of the migrated data sets. Hence, recall is not a factor in the time it takes to execute the statement.

Dropping a trusted context:
The drop of a trusted context takes effect after the DROP TRUSTED CONTEXT statement is committed. If the DROP TRUSTED CONTEXT statement results in an error or is rolled back, the trusted context is not dropped.

Deleting SYSLGRNG records for dropped table spaces:
After dropping a table space, you cannot delete the associated records. If you want to remove the records, you must quiesce the table space, and then run the MODIFY RECOVERY utility *before* dropping the table space. If you delete the SYSLGRNG records and drop the table space, you cannot reclaim the table space.

Invalidation of packages and dynamic cached statements after dropping row permissions or column masks:
If row or column access control is currently enforced for the table, dropping the row permission or the column mask invalidates all packages and dynamic cached statements that reference the table. Otherwise no package or dynamic cached statement is invalidated.

Dependencies when dropping objects:
Whenever an object is directly or indirectly dropped, other objects that depend on the dropped object might also be dropped. (The catalog stores information about the dependencies of objects on each other.) The following semantics determine what happens to a dependent object when the object that it depends on (the underlying object) is dropped:

Cascade (D)
Dropping the underlying object causes the dependent object to be dropped. However, if the dependent object cannot be dropped because it has a restrict dependency on another object, the drop of the underlying object fails.

Restrict (D)
The underlying object cannot be dropped if a dependent object exists.

Inoperative (O)
Dropping the underlying object causes the dependent object to become inoperative.

Invalidation (V)
Dropping the underlying object causes the dependent object to become invalidated.

For objects that directly depend on others, the following table uses the letter abbreviations above to summarize what happens to a dependent object when its underlying object is specified in a DROP statement. Additional objects can be indirectly affected, too.

To determine the indirect effects of a DROP statement, assess what happens to the dependent object and whether the dependent object has objects that depend on it. For example, assume that view B is defined on table A and view C is defined on view B. In the following table, the 'D' in the VIEW column of the DROP TABLE row indicates that view B is dropped when table A is dropped. Next, because view C is dependent on view B, check the VIEW column for DROP VIEW. The 'D' in the column indicates that view C will be dropped, too.

The letters in the following table have the following meanings:

D Dependent object is dropped.
O Dependent object is made inoperative.
V Dependent object is invalidated.
R DROP statement fails.

Table 138. Effect of dropping objects that have dependencies

	Type of object														
DROP statement	**Alias**	**Database**	**Function**	**Global variable**	**Index**	**Package**[1]	**Procedure**	**Sequence**	**Stogroup**	**Synonym**	**Table**	**Tablespace**	**Trigger**	**Type**	**View**
DROP ALIAS						V									
DROP DATABASE					D[2]						D	D			D
DROP FUNCTION			R[5] 17 18			O	R[17] 18						R		R
DROP INDEX[2,6]						V							V		
DROP PACKAGE[7]															
DROP PROCEDURE			R 17 18			O	R[17] 18						R		
DROP ROLE						V									
DROP SEQUENCE			R[13]			V							R		
DROP STOGROUP					R[8]							R[8]			
DROP SYNONYM															
DROP TABLE[9,10]					D	V				D			D[11]		D
DROP TABLESPACE[12]					D	V					D				

Table 138. Effect of dropping objects that have dependencies (continued)

	Type of object														
DROP statement	**Alias**	**Database**	**Function**	**Global variable**	**Index**	**Package**[1]	**Procedure**	**Sequence**	**Stogroup**	**Synonym**	**Table**	**Tablespace**	**Trigger**	**Type**	**View**
DROP TRIGGER						V[16]									
DROP TYPE			R[3]		R[14]		R[4]	R			R				
DROP VARIABLE			R			V							R		R
DROP VIEW						V				D			D[15]		D

Notes:

1. The PACKAGE column represents packages for user-defined functions, procedures, and triggers, as well as other packages. The PACKAGE column also applies for plans.
2. The index space associated with the index is dropped.
3. If a function is dependent on the user-defined type being dropped, the user-defined type cannot be dropped unless the function is one of the cast functions that was created for the user-defined type.
4. If the definition of a parameter of a stored procedure uses the user-defined type, the user-defined type cannot be dropped.
5. If other user-defined functions are sourced on the user-defined function being dropped, the function cannot be dropped.
6. An index on an auxiliary table cannot be explicitly dropped.
7. A trigger package cannot be explicitly dropped with DROP PACKAGE. A trigger package is implicitly dropped when the associated trigger or subject table is dropped.
8. A storage group cannot be dropped if it is used by any table space or index space.
9. An auxiliary table cannot be explicitly dropped with DROP TABLE. An auxiliary table is implicitly dropped when the associated base table is dropped.
10. If an implicit table space was created when the table was created, the table space is also dropped.
11. When a subject table is dropped, any associated triggers and related trigger packages are also dropped.
12. A LOB table space cannot be dropped until the base table with the LOB columns is dropped.
13. This restriction is only for SQL functions.
14. The index in this case must be an expression-based index.
15. When a subject view is dropped, any associated triggers and related trigger packages are also dropped.
16. Any packages that have a dependency on an INSTEAD OF trigger will be marked invalid.
17. A routine that is referenced by a non-inline SQL scalar function cannot be dropped.
18. A routine that is referenced by a native SQL procedure cannot be dropped.

Alternative syntax and synonyms:

To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords:

- DATA TYPE or DISTINCT TYPE as a synonym for TYPE
- PROGRAM as a synonym for PACKAGE
- DROP ALIAS SYSPUBLIC.*name* can be specified as an alternative to DROP PUBLIC ALIAS SYSPUBLIC.*name*

Examples

Example 1: Drop table DSN8B10.DEPT.

```
DROP TABLE DSN8B10.DEPT;
```

Example 2: Drop table space DSN8S11D in database DSN8D11A.

```
DROP TABLESPACE DSN8D11A.DSN8S11D;
```

Example 3: Drop the view DSN8B10.VPROJRE1:

```
DROP VIEW DSN8B10.VPROJRE1;
```

Example 4: Drop the package DSN8CC0 with the version identifier VERSZZZZ. The package is in the collection DSN8CC61. Use the version identifier to distinguish the package to be dropped from another package with the same name in the same collection.

```
DROP PACKAGE DSN8CC61.DSN8CC0 VERSION VERSZZZZ;
```

Example 5: Drop the package DSN8CC0 with the version identifier "1994-07-14-09.56.30.196952". When a version identifier is generated by the VERSION(AUTO) precompiler option, delimit the version identifier.

```
DROP PACKAGE DSN8CC61.DSN8CC0 VERSION "1994-07-14-09.56.30.196952";
```

Example 6: Drop the distinct type DOCUMENT, if it is not currently in use:

```
DROP TYPE DOCUMENT;
```

Example 7: Assume that you are SMITH and that ATOMIC_WEIGHT is the only function with that name in schema CHEM. Drop ATOMIC_WEIGHT.

```
DROP FUNCTION CHEM.ATOMIC_WEIGHT;
```

Example 8: Assume that you are SMITH and that you created the function CENTER in schema SMITH. Drop CENTER, using the function signature to identify the function instance to be dropped.

```
DROP FUNCTION CENTER(INTEGER, FLOAT);
```

Example 9: Assume that you are SMITH and that you created another function named CENTER, which you gave the specific name FOCUS97, in schema JOHNSON. Drop CENTER, using the specific name to identify the function instance to be dropped.

```
DROP SPECIFIC FUNCTION JOHNSON.FOCUS97;
```

Example 10: Assume that you are SMITH and that stored procedure OSMOSIS is in schema BIOLOGY. Drop OSMOSIS.

```
DROP PROCEDURE BIOLOGY.OSMOSIS;
```

Example 11: Assume that you are SMITH and that trigger BONUS is in your schema. Drop BONUS.

```
DROP TRIGGER BONUS;
```

Example 12: Drop the role CTXROLE:

```
DROP ROLE CTXROLE;
```

Example 13: Drop the trusted context CTX1:

```
DROP TRUSTED CONTEXT CTX1;
```

END DECLARE SECTION

The END DECLARE SECTION statement marks the end of an SQL declare section.

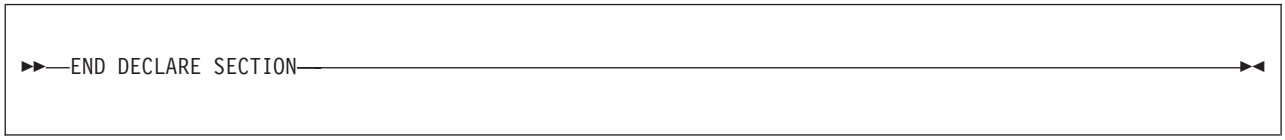
Invocation

This statement can only be embedded in an application program. It is not an executable statement. It must not be specified in Java or REXX.

Authorization

None required.

Syntax



Description

The END DECLARE SECTION statement can be coded in the application program wherever declarations can appear in accordance with the rules of the host language. It is used to indicate the end of an SQL declare section. An SQL declare section starts with a BEGIN DECLARE SECTION statement described in "BEGIN DECLARE SECTION" on page 1115.

The following rules are enforced by the precompiler only if the host language is C or the STDSQL(YES) precompiler option is specified:

- A variable referred to in an SQL statement must be declared within an SQL declare section of the source program.
- BEGIN DECLARE SECTION and END DECLARE SECTION statements must be paired and must not be nested.
- SQL declare sections can contain only host variable declarations, SQL INCLUDE statements that include host variable declarations, or DECLARE VARIABLE statements.

Notes

SQL declare sections are only required if the STDSQL(YES) option is specified or the host language is C. However, SQL declare sections can be specified for any host language so that the source program can conform to IBM SQL. If SQL declare sections are used, but not required, variables declared outside an SQL declare section should not have the same name as variables declared within an SQL declare section.

Example

```
EXEC SQL BEGIN DECLARE SECTION;

 -- host variable declarations

EXEC SQL END DECLARE SECTION;
```

EXCHANGE

The EXCHANGE statement switches the content of a base table and its associated clone table.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

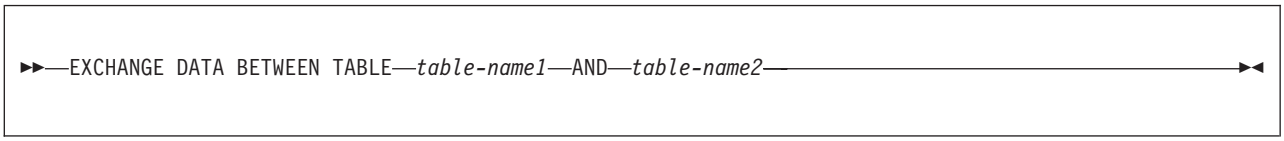
Authorization

The privilege set that is defined below must include at least one of the following privileges:

- The INSERT and DELETE privileges on both the base table and the clone table
- Ownership of the both the base table and the clone table
- DBADM authority for the database
- SYSADM authority
- DATAACCESS authority

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set is the union of the privilege sets that are held by each authorization ID of the process.

Syntax



Description

table-name1 **and** *table-name2*
Identifies the base table and the associated clone table for which the exchange of data will take place. Either *table-name1* or *table-name2* can identify the base table. The other table name must identify a clone table that is associated with the specified base table. The name of the base table and the name of the clone table remain unchanged after a data exchange.

Notes

Rules and restrictions: Data exchanges cannot be done for a subset of a table's partitions. There must be a commit between consecutive data exchanges using the EXCHANGE statement.

Examples

Example: Exchange the data of the EMPLOYEE table and its clone table, EMPCLONE.

```
EXCHANGE DATA BETWEEN TABLE EMPCLONE AND EMPLOYEE;
```

EXECUTE

The EXECUTE statement executes a prepared SQL statement.

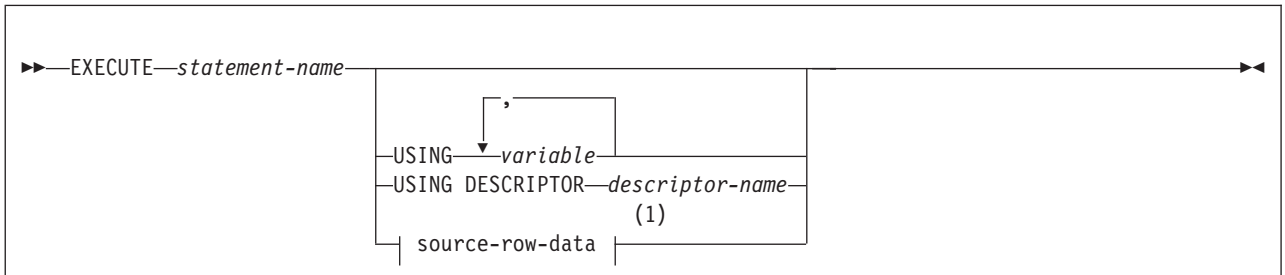
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

See "PREPARE" on page 1781 for the authorization required to create a prepared statement.

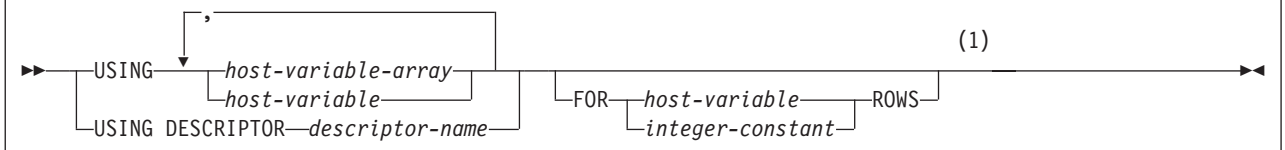
Syntax



Notes:

1 This option can be specified only when *statement-name* refers to a dynamic INSERT or MERGE statement that is prepared with FOR MULTIPLE ROWS and is specified as part of the ATTRIBUTES clause on the PREPARE statement.

source-row-data:



Notes:

1 The FOR n ROWS clause is required on the EXECUTE statement if it is not specified as part of the MERGE statement and a host variable array is specified. The FOR n ROWS clause is also required if MERGE is used with multiple rows of source data. For an INSERT statement, the FOR n ROWS clause can only be specified for a dynamic statement that contains only a single multiple-row INSERT statement.

Description

statement-name
Identifies the prepared statement to be executed. *statement-name* must identify a

statement that was previously prepared within the unit of work and the prepared statement must not be a SELECT statement.

USING
Introduces a list of variables whose values are substituted for the parameter markers (question marks) in the prepared statement. (For an explanation of parameter markers, see "PREPARE" on page 1781.) If the prepared statement includes parameter markers, you must include USING in the EXECUTE statement. USING is ignored if there are no parameter markers.

For more on the substitution of values for parameter markers, see Parameter marker replacement.

variable,...
Identifies a variable or a host structure that is declared in the application program in accordance with the rules for declaring variables and host structures. When the statement is executed, a reference to a structure is replaced by a reference to each of its variables. The number of variables must be the same as the number of parameter markers in the prepared statement. The *n*th variable corresponds to the *n*th parameter marker in the prepared statement. Where appropriate, locator variables and file reference variables can be provided as the source of values for parameter markers.

USING DESCRIPTOR *descriptor-name*
Identifies an SQLDA that contains a valid description of the input host variables.

Before invoking the EXECUTE statement, you must set the following fields in the SQLDA:

- SQLN to indicate the number of SQLVAR occurrences that are provided in the SQLDA

  A REXX SQLDA does not contain this field.
- SQLABC to indicate the number of bytes of storage that are allocated for the SQLDA
- SQLD to indicate the number of variables that are used in the SQLDA when processing the statement
- SQLVAR entries to indicate the attributes of the variables

The SQLDA must have enough storage to contain all SQLVAR entries. If an SQLVAR entry includes a LOB value or a distinct type based on a LOB, there must be additional SQLVAR entries for each parameter. For more information on the SQLDA, which includes a description of the SQLVAR and an explanation on how to determine the number of SQLVAR entries, see "SQL descriptor area (SQLDA)" on page 2079.

SQLD must be set to a value that is greater than or equal to zero and less than or equal to SQLN. It must be the same as the number of parameter markers in the prepared statement. The *n*th variable described by the SQLDA corresponds to the *n*th parameter marker in the prepared statement.

See "Identifying an SQLDA in C or C++" on page 2099 for how to represent *descriptor-name* in C.

source-row-data
The prepared statement must be an INSERT or MERGE statement for which the FOR MULTIPLE ROWS clause is specified as part of the ATTRIBUTES clause on the PREPARE statement.

USING *host-variable-array* **or** *host-variable*
Introduces a list of host variables or host variable arrays whose values are

substituted for the parameter markers (question marks) in the prepared INSERT or MERGE statement. The number of columns specified in the INSERT or MERGE statement must be less than or equal to the total number of host variables or host variable arrays that are specified.

host-variable-array
Identifies a host-variable array that must be defined in the application program in accordance with the rules for declaring a host variable array. A reference to a structure is replaced by a reference to each of its variables. The number of variables must be the same as the number of parameter markers in the prepared statement. The *n*th variable supplies the value for the *n*th parameter marker in the prepared statement.

host-variable
Identifies a variable that must be described in the application program in accordance with the rules for declaring host variables.

USING DESCRIPTOR *descriptor-name*
Identifies an SQLDA that must contain a valid description of the host variable arrays or host variables that contain the values to insert.

Before invoking the EXECUTE statement for a dynamic INSERT or MERGE statement, you must set the following fields in the SQLDA:

- SQLN to indicate the number of SQLVAR entries that are provided in the SQLDA.
- SQLABC to indicate the number of bytes of storage that are allocated for the SQLDA.
- SQLD to indicate the number of variables, plus one, that are used in the SQLDA that provide values for columns that are the source of the INSERT or MERGE statement. SQLD must be set to a value that is greater than or equal to zero and less than or equal to SQLN.
- SQLVAR entries to indicate the attributes of an element of the host variable array for the SQLVAR entries that correspond to values that are provided for the source columns of the INSERT or MERGE statement. Within each SQLVAR, the following fields are set:
  - SQLTYPE indicates the data type of the elements of the host variable array.
  - SQLDATA points to the corresponding host variable array.
  - SQLLEN and SQLLONGLEN indicate the length of a single element of the array.
- SQLNAME, the fifth and sixth bytes must contain a flag field and the seventh and eighth bytes must contain a binary small integer (halfword) that contains the dimension of the host-variable array and, if specified, the corresponding indicator array.

The SQLDA must have enough storage to contain a SQLVAR entry for each target column for which values are provided, plus an additional SQLVAR entry for the number of rows. The DB2 system generates code to enter the required information for this extra SQLVAR entry. Each SQLVAR entry describes a host variable, host variable array, or buffer that contains the values for a column of the source table. The last SQLVAR entry contains the number of rows of data. For example, if the INSERT or MERGE statement is providing values for five columns of the target table, six SQLVAR entries must be provided. If any value is a LOB value, twice as many SQLVAR entries must be provided, and SQLN must be set to the number of SQLVAR entries. Thus, if the INSERT or MERGE

statement is providing values for five columns of the source table, and some of the values to insert are LOB values, 12 SQLVAR entries must be provided.

The SQLVAR entry for the number of rows must also contain a flag value. See "Field descriptions of an occurrence of a base SQLVAR" on page 2085 for more information.

You set the SQLDATA and SQLIND pointers to the beginning of the corresponding arrays.

FOR *host-variable* **or** *integer-constant* **ROWS**
Specifies the number of rows of source data. The values for the insert or merge operation are specified in the USING clause.

host-variable or *integer-constant* is assigned to an integral value *k*. If *host-variable* is specified, it must be an exact numeric type with a scale of zero and must not include an indicator variable. *k* must be in the range 0 to 32767.

FOR n ROWS cannot be specified on the EXECUTE statement if the statement being processed is a dynamic INSERT or MERGE statement that includes a FOR n ROWS clause.

Notes

Excessive processor time:
DB2 can stop the execution of a prepared SQL statement if the statement is taking too much processor time to finish. When this happens, an error occurs. The application that issued the statement is not terminated; it is allowed to issue another SQL statement.

Parameter marker replacement:
Before the prepared statement is executed, each parameter marker in the statement is effectively replaced by its corresponding host variable. The replacement is an assignment operation in which the source is the value of the host variable and the target is a variable within DB2. The assignment rules are those described for assignment to a column in "Assignment and comparison" on page 121. For a typed parameter marker, the attributes of the target variable are those specified by the CAST specification. For an untyped parameter marker, the attributes of the target variable are determined according to the context of the parameter marker. For the rules that affect parameter markers, see Parameter markers.

Let V denote a host variable that corresponds to parameter marker P. The value of V is assigned to the target variable for P in accordance with the rules for assigning a value to a column:

- V must be compatible with the target.
- If V is a string, its length must not be greater than the length attribute of the target.
- If V is a number, the absolute value of its integral part must not be greater than the maximum absolute value of the integral part of the target.
- If the attributes of V are not identical to the attributes of the target, the value is converted to conform to the attributes of the target.
- If the target cannot contain nulls, V must not be null.

When the prepared statement is executed, the value used in place of P is the value of the target variable for P. For example, if V is CHAR(6) and the target is CHAR(8), the value used in place of P is the value of V padded on the right with two blanks.

Errors occurring on EXECUTE:

In local and remote processing, the DEFER(PREPARE) and REOPT(ALWAYS)/REOPT(ONCE) bind options can cause some errors that are normally issued during PREPARE processing to be issued on EXECUTE.

Considerations for executing data definition statements written in native SQL language:

A data definition statement written in native SQL language can only be executed one time. To execute the data definition statement multiple times, issue the PREPARE statement prior to each use of the EXECUTE statement for the data definition statement.

Examples

Example 1: In this example, an INSERT statement with parameter markers is prepared and executed. S1 is a structure that corresponds to the format of DSN8B10.DEPT.

```
EXEC SQL PREPARE DEPT_INSERT FROM
  'INSERT INTO DSN8B10.DEPT VALUES(?,?,?,?)';
-- Check for successful execution and read values into S1
EXEC SQL EXECUTE DEPT_INSERT USING :S1;
```

Example 2: Assume that the IWH.PROGPARM table has 9 columns. Prepare and execute a dynamic INSERT statement that inserts 5 rows of data into the IWH.PROGPARM table. The values to be inserted are provided in arrays, where all the values for a column are provided in an host-variable-array with the EXECUTE statement.

```
STMT = 'INSERT INTO IWH.PROGPARM  (IWHID, UPDATE_BY,UPDATE_TS,NAME,
                                   SHORT_DESCRIPTION, ORDERNO, PARMDATA,
                                   PARMDATALONG, VWPROGKEY)
VALUES ( ? , ? , ? , ? , ? , ? , ? , ? , ? )';
ATTRVAR = 'FOR MULTIPLE ROWS';
EXEC SQL PREPARE INS_STMT ATTRIBUTES :ATTRVAR FROM :STMT;
NROWS = 5;
EXEC SQL EXECUTE INS_STMT FOR :NROWS ROWS
           USING :V1, :V2, :V3, :V4, :V5, :V6, :V7, :V8, :V9;
```

In this example, each host variable in the USING clause represents an array of values for the corresponding column of the target of the INSERT statement.

Example 3: Using dynamically supplied values for an employee row, update the master EMPLOYEE table if the data is for an existing employee or insert a new row if the data is for a new employee.

```
hv_stmt =
  "MERGE INTO EMPLOYEE AS T
    USING (VALUES (CAST (? AS CHAR(6)), CAST (? AS VARCHAR(12)),
                   CAST (? AS CHAR(1)), CAST (? AS VARCHAR(15)),
                   CAST (? AS INTEGER)))
    AS S (EMPNO, FIRSTNAME, MI, LASTNAME, SALARY)
    ON T.EMPNO = S.EMPNO
    WHEN MATCHED THEN UPDATE
        SET SALARY = S.SALARY
    WHEN NOT MATCHED THEN INSERT (EMPNO, FIRSTNAME, MI, LASTNAME, SALARY)
        VALUES (S.EMPNO, S.FIRSTNAME, S.MI, S.LASTNAME, S.SALARY)
    NOT ATOMIC CONTINUE ON SQLEXCEPTION";
hv_attr = 'FOR MULTIPLE ROWS';
EXEC SQL
    PREPARE merge_stmt
     ATTRIBUTES :hv_attr FROM :hv_stmt;
```

```
hv_nrows = 5;
/* Initialize the hostvar array of hv_empno, hv_firstname... */
EXEC SQL
   EXECUTE merge_stmt
   USING :hv_empno, :hv_firstname, :hv_mi,
         :hv_lastname, :hv_salary
      FOR :hv_nrows ROWS;
```

Example 4: Suppose that the following array type, array variable, and table have been defined.

```
CREATE TYPE INTARRAY AS INTEGER ARRAY[100];
CREATE TYPE STRINGARRAY AS VARCHAR(10) ARRAY[100];
CREATE TABLE T1 (COL1 CHAR(10), COL2 INT);
```

Use as an array variable as an input value for an expression in an EXECUTE statement.

```
CREATE PROCEDURE PROCESSPERSONS (OUT WITHO STRINGARRAY, INOUT INT0 INT)
BEGIN
 DECLARE INTA INTARRAY;
 DECLARE STMT CHAR(100);
-- Initialize the array
 SET INTA = ARRAY[1,INTEGER(2),3+0,4,5,6] ;
-- Use dynamic sql with an array parameter marker to
--  provide a value for a dynamic INSERT statement
 SET STMT = 'INSERT INTO T1 VALUES('XYZ', CARDINALITY(CAST(? AS INTARRAY)))';
 PREPARE INS_STMT FROM STMT;
 EXECUTE INS_STMT USING INTA;
-- INTA is an array variable used as input for the
-- INSERT statement
...
END
```

EXECUTE IMMEDIATE

EXECUTE IMMEDIATE combines the basic functions of the PREPARE and EXECUTE statements. It can be used to prepare and execute an SQL statement that contains neither host variables nor parameter markers.

The EXECUTE IMMEDIATE statement:

- Prepares an executable form of an SQL statement from a string form of the statement
- Executes the SQL statement
- Destroys the executable form

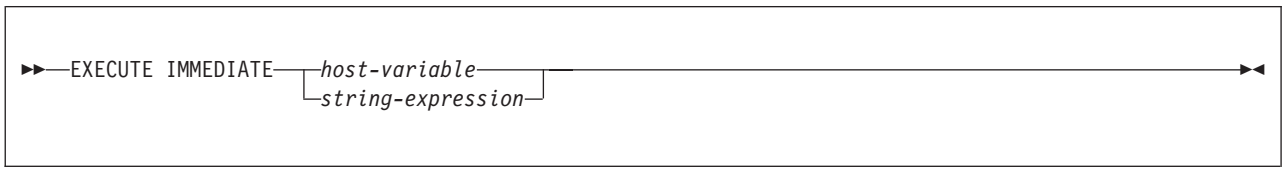
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

The authorization rules are those defined for the dynamic preparation of the SQL statement specified by EXECUTE IMMEDIATE. For example, see "INSERT" on page 1734 for the authorization rules that apply when an INSERT statement is executed using EXECUTE IMMEDIATE.

Syntax



Description

host-variable

host-variable must be specified. It must identify a host variable that is described in the application program in accordance with the rules for declaring character or graphic string variables. If the source string is over 32KB in length, the *host-variable* must be a CLOB or DBCLOB variable. The maximum source length is 2MB although the host variable can be declared larger than 2MB. An indicator variable must not be specified. In Assembler, C, COBOL, and PL/I, the host variable must be a varying-length string variable. In C, it must not be a NUL-terminated string. In SQL PL, an SQL variable is used in place of a host variable and the value must not be null.

string-expression

string-expression is any PL/I expression that yields a string. *string-expression* cannot be preceded by a colon. Variables that are within *string-expression* that include operators or functions should not be preceded by a colon. When *string-expression* is specified, the precompiler-generated structures for *string-expression* use an EBCDIC CCSID and an informational message is returned.

Notes

Allowable SQL statements:

The value of the identified host variable or the specified *string-expression* is called the *statement string*.

The statement string must be one of the following SQL statements, and cannot be the *select-statement*:

ALLOCATE CURSOR	REVOKE
ALTER	ROLLBACK
ASSOCIATE LOCATORS	SAVEPOINT
COMMENT	SET CURRENT DEGREE
COMMIT	SET CURRENT DECFLOAT ROUNDING MODE
CREATE	SET CURRENT DEBUG MODE
DECLARE GLOBAL TEMPORARY TABLE	SET CURRENT LOCALE LC_CTYPE
DELETE	SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION
DROP	SET CURRENT OPTIMIZATION HINT
EXPLAIN	SET CURRENT PRECISION
FREE LOCATOR	SET CURRENT QUERY ACCELERATION
GRANT	SET CURRENT REFRESH AGE
HOLD LOCATOR	SET CURRENT ROUTINE VERSION
INSERT	SET CURRENT RULES
LABEL	SET CURRENT SQLID
LOCK TABLE	SET ENCRYPTION PASSWORD
MERGE	SET PATH
REFRESH TABLE	SET SCHEMA
RELEASE SAVEPOINT	SIGNAL
RENAME	TRUNCATE
	UPDATE

The statement string must not:

- Begin with EXEC SQL
- End with END-EXEC or a semicolon
- Include references to variables
- Include parameter markers

Errors and error handling:

When an EXECUTE IMMEDIATE statement is executed, the specified statement string is parsed and checked for errors. If the SQL statement is invalid, it is not executed and the error condition that prevents its execution is reported in the SQLCA. If the SQL statement is valid, but an error occurs during its execution, that error condition is reported in the SQLCA.

DB2 can stop the execution of a prepared SQL statement if the statement is taking too much CPU time to finish. When this happens an error occurs. The application that issued the statement is not terminated; it is allowed to issue another SQL statement.

Effect of the CURRENT EXPLAIN MODE special register:

If the CURRENT EXPLAIN MODE special register is set to EXPLAIN, the statement is prepared for explain only and is not executable, unless the statement is a SET statement. Attempting to execute the prepared statement will return an error. See the "CURRENT EXPLAIN MODE" on page 175 special register for more information.

Performance considerations:

If the same SQL statement is to be executed more than once, it is more efficient to use the PREPARE and EXECUTE statements rather than the EXECUTE IMMEDIATE statement.

Examples

Example 1: In this PL/I example, the EXECUTE IMMEDIATE statement is used to execute a DELETE statement in which the rows to be deleted are determined by a search-condition specified by the value of PREDS.

```
EXEC SQL EXECUTE IMMEDIATE 'DELETE FROM DSN8B10.DEPT
  WHERE' || PREDS;
```

Example 2: Use C to execute the SQL statement in the host variable Qstring.

```
EXEC SQL INCLUDE SQLCA;
void main ()
  {
   EXEC SQL BEGINDECLARE SECTION;
   char Qstring[100M =
     "INSERT INTO WORK_TABLE SELECT * FROM EMPPROJACT WHERE ACTNO >= 100";
   EXEC SQL END DECLARE SECTION;
  .
  .
  .
   EXEC SQL EXECUTE IMMEDIATE :Qstring;
   return;
  }
```

EXPLAIN

The EXPLAIN statement obtains information about access path selection for an *explainable statement*. A statement is explainable if it is a SELECT, MERGE, or INSERT statement, or the searched form of an UPDATE or DELETE statement. The information that is obtained is placed in a set of supplied user tables that are called *EXPLAIN tables*.

The *plan table* contains information about the access path for the explained statement. The *statement table* can be populated with information about the estimated cost of executing the explainable statement. The *function table* can be populated with information about how DB2 resolves the user-defined functions that are referred to in the explainable statement. Other EXPLAIN tables can be populated with additional information about the execution of the explainable statement. For a complete list of EXPLAIN tables, see "EXPLAIN tables" on page 2470.

Using EXPLAIN in queries that references system-period temporal tables that are enabled for system data versioning, the result will show the system-period temporal tables and the history tables in EXPLAIN output if the query needs to reference both tables to satisfy the query.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The authorization rules are those defined for the SQL statement specified in the EXPLAIN statement. For example, see the description of the DELETE statement for the authorization rules that apply when a DELETE statement is explained.

If the EXPLAIN statement is embedded in an application program, the authorization rules that apply are those defined for embedding the specified SQL statement in an application program. In addition, the owner of the plan or package must also have one of the following characteristics:

- Be the owner of a plan table named PLAN_TABLE
- Have an alias on a plan table named *owner*.PLAN_TABLE and have SELECT and INSERT privileges on the table

If the EXPLAIN statement is dynamically prepared, the authorization rules that apply are those defined for dynamically preparing the specified SQL statement. In addition, the SQL authorization ID of the process or the role this is associated with the process (if the EXPLAIN statement is running in a trusted context that specifies the ROLE AS OBJECT OWNER AND QUALIFIER clause) must also have one of the following characteristics:

- Be the owner of a plan table named PLAN_TABLE
- Have an alias on a plan table named *owner*.PLAN_TABLE and have SELECT and INSERT privileges on the table

To issue the EXPLAIN statement with the PLAN and ALL keywords, the privilege set that is defined below must include at least one of the following:

- EXPLAIN
- SQLADM
- System DBADM
- The authorization rules that are defined for the SQL statement specified in the EXPLAIN statement. For example, the authorization rules that apply when a DELETE statement is explained are the authorization rules for the DELETE statement.

The authorization rules are different if the STMTCACHE keyword is specified to have a cached statement explained. The privilege set must include at least one of the following:

- SQLADM authority
- SYSADM authority
- The authority that is required to share the cached statement.
- System DBADM authority

For the STMTCACHE ALL keyword, the privilege set must include at least one of the following:

- SQLADM authority
- System DBADM authority
- SYSADM authority to explain all statements in the dynamic statement cache

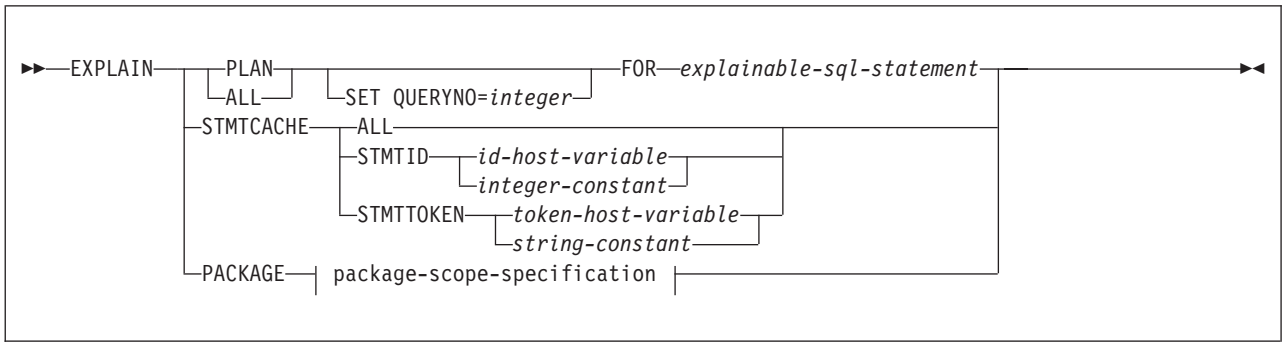
If the privilege set does not have the required authority, only those statements that have the same authorization ID as the privilege set are explained.

For the PACKAGE keyword, the privilege set must include at least one of the following:

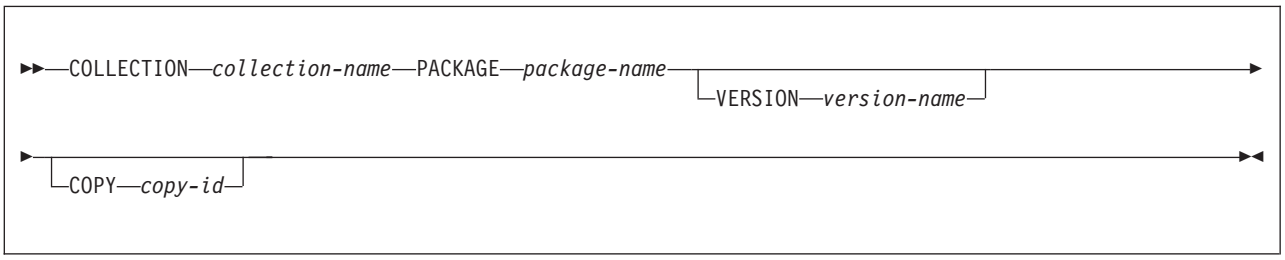
- SQLADM authority
- SYSADM authority
- SYSOPR authority
- SYSCTRL authority

Privilege set: The privilege set comprises the union of authorities that are held by the authorization IDs of the process. If the process is running in a trusted context with a role, this role would be included as an authorization ID of the process.

Syntax



package-scope-specification:



Description

PLAN

Inserts one row into the plan table for each step used in executing *explainable-sql-statement*. The steps for enforcing referential constraints are not included.

If a statement table exists, one row that provides a cost estimate of processing the explainable statement is inserted into the statement table. If the explainable statement is a SELECT FROM *data-change-statement*, two rows are inserted into the statement table.

If a function table exists, one row is inserted into the function table for each user-defined function that is referred to by the explainable statement.

If additional EXPLAIN tables exist, rows are also inserted into those tables.

Related reference:

"EXPLAIN tables" on page 2470

"PLAN_TABLE" on page 2471

"DSN_STATEMNT_TABLE" on page 2573

"DSN_FUNCTION_TABLE" on page 2509

ALL

Has the same effect as PLAN.

SET QUERYNO = *integer*

Associates *integer* with *explainable-sql-statement*. The column QUERYNO is given the value *integer* in every row inserted into the plan table, statement table, or function table by the EXPLAIN statement. If QUERYNO is not specified, DB2 itself assigns a number. For an embedded EXPLAIN statement, the number is the statement number that was assigned by the precompiler and placed in the DBRM.

FOR *explainable-sql-statement*

Specifies the SQL statement to be explained. *explainable-sql-statement* can be any explainable SQL statement. If EXPLAIN is embedded in a program, the statement can contain references to host variables. If EXPLAIN is dynamically prepared, the statement can contain parameter markers. Host variables that appear in the statement must be defined in the statement's program.

The statement must refer to objects at the current server.

explainable-sql-statement must not contain a QUERYNO clause. To specify the value of the QUERYNO column in plan table for the statement being explained, use the SET QUERYNO = clause of the EXPLAIN statement.

explainable-sql-statement cannot be a statement-name or a host-variable. To use EXPLAIN to get information about dynamic SQL statements, you must prepare the entire EXPLAIN statement dynamically.

To obtain information about an explainable SQL statement that references a declared temporary table, the EXPLAIN statement must be executed in the same application process in which the table was declared. For static EXPLAIN statements, the information is not obtained at bind-time but at run time when the EXPLAIN statement is incrementally bound.

STMTCACHE
Specifies that statements in the dynamic statement cache are to be explained. In a data sharing environment, the statements in the dynamic statement cache of the data sharing member where EXPLAIN STMTCACHE is executed are explained.

ALL
Specifies that all of the cached statements are to be explained. STMTCACHE ALL returns one row for each cached statement to the DSN_STATEMENT_CACHE_TABLE. These rows contain identifying information about the statements in the cache, as well as statistics that reflect the execution of the statements by all processes that have executed the statement. Records are not returned to other EXPLAIN tables when STMTCACHE ALL is specified.

STMTID *id-host-variable* **or** *integer-constant*
Specifies that the cached statement with the specified statement ID is to be explained. The value contained in *id-host-variable* or specified by *integer-constant* identifies the statement ID. STMTCACHE STMTID returns rows to the following EXPLAIN tables:

- PLAN_TABLE
- DSN_STATEMNT_TABLE
- DSN_FUNCTION_TABLE
- DSN_STATEMENT_CACHE_TABLE

The statement ID is an integer that uniquely identifies a statement that has been cached in the dynamic statement cache. The statement ID of a cached statement can be retrieved through IFI monitor facilities from IFCID 0316 or 0124. Some diagnostic trace records, such as IFCIDs 0173, 0196, and 0337, also show the statement ID.

The QUERYNO column of each EXPLAIN table record that is returned contains the statement ID value.

STMTTOKEN *id-host-variable* **or** *string-constant*
Specifies that the cached statements with the specified statement token are to be explained. The value contained in *token-host-variable* or specified by *string-constant* identifies the statement token. STMTCACHE STMTTOKEN returns rows to the following EXPLAIN tables:

- PLAN_TABLE
- DSN_STATEMNT_TABLE
- DSN_FUNCTION_TABLE
- DSN_STATEMENT_CACHE_TABLE

The statement token must be a character string that is no longer than 240 bytes. The application program that originally prepares and inserts a statement into the cache associates a statement token with the cached

statement. The program can make this association with the RRSAF SET_ID function, or the sqleseti API if the program is connected remotely.

The STMTTOKEN column of each PLAN_TABLE record that is returned contains the statement token value. The QUERYNO column of each EXPLAIN table record that is returned contains the statement ID value.

Related information:

"DSN_STATEMENT_CACHE_TABLE" on page 2567

PACKAGE

Specifies that all static SQL statements in the package that matches the specified scope are explained. The explain information is added to the PLAN_TABLE that is owned by the current user. Packages that are bound prior to DB2 Version 9.1 for z/OS are not explained.

COLLECTION *collection-name*

Specifies that only statements under the specified *collection-name* are to be explained. *collection-name* is a string constant or a host variable that represents the collection name.

PACKAGE *package-name*

Specifies that only statements under the specified *package-name* are to be explained. *package-name* is a string constant or a host variable that represents the package name.

VERSION *version-name*

Specifies that only statements under the specified *version-name* are to be explained. *version-name* is a string constant or a host variable that represents the version name.

COPY *copy-id*

Specifies that only statements under the specified *copy-id* are to be explained. *copy-id* must be one of the following values:

- CURRENT
- PREVIOUS
- ORIGINAL

If the COPYID clause is not specified, statements under all copies that exist for that package (CURRENT, PREVIOUS, and ORIGINAL) are explained.

The HINT_USED column in the PLAN_TABLE is populated with `EXPLAIN PACKAGE: copy-id`. *copy-id* in the HINT_USED column will be one of the following values:

- "0" - the current copy
- "1" - the previous copy
- "2" - the original copy

Notes

Output from EXPLAIN:

DB2 inserts one or more rows of data into a plan table and other existing *EXPLAIN tables*. A plan table must exist before the operation that results in EXPLAIN output. For information about valid plan table formats see "PLAN_TABLE" on page 2471. You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Unless you need the information that is provided by the additional EXPLAIN tables, it is not necessary to create those tables to use EXPLAIN. However, a statement cache table is required when the STMTCACHE ALL keyword is specified as part of an EXPLAIN statement.

Column access control or row permissions enforced for EXPLAIN tables:
Column access control and row permissions can be enforced for EXPLAIN tables. However, row permissions and column masks are not applied when DB2 inserts rows into those tables.

If the statement to be explained references tables for which row or column access control is activated, the following information from row permission and column mask definitions created for the tables might appear in the EXPLAIN tables:

- DSN_FUNCTION_TABLE - user-defined functions
- DSN_PREDICAT_TABLE - predicates (except predicates in CASE WHEN clauses)
- DSN_STRUCT_TABLE - query blocks
- PLAN_TABLE - access path of subqueries

In addition, the complete or partial definition text might appear in EXPLAIN tables like DSN_FUNCTION_TABLE, DSN_PREDICAT_TABLE, DSN_QUERY_TABLE, DSN_SORTKEY_TABLE, DSN_STATEMENT_CACHE_TABLE, and DSN_STATEMENT_RUNTIME_INFO.

Impact to the existing access paths when the table has enforced column access control or row permissions:
The predicates from the row permissions are considered in the access path selection. Therefore, they are shown in the EXPLAIN tables for the performance tuning purpose.

EXPLAIN tables:
Each row in an EXPLAIN table describes some aspect of a step in the execution of a query or subquery in an explainable statement. The column values for the row identify, among other things, the query or subquery, the tables and other objects involved, the methods used to carry out each step, and cost information about those methods.

Instances of these tables might also be created and used by certain optimization tools. For information about the meanings of different values in plan table and other EXPLAIN tables, see .

For information about how to correlate information across EXPLAIN tables, see Correlating information across EXPLAIN tables (DB2 Performance).

Important: Do not manually manipulate the data in EXPLAIN tables that are created by optimization tools.

Plan table
owner.PLAN_TABLE.

Function table
owner.DSN_FUNCTION_TABLE.

Statement table
owner.DSN_STATEMNT_TABLE.

Statement cache table
owner.DSN_STATEMENT_CACHE_TABLE.

Structure table
owner.DSN_STRUCT_TABLE.

Predicate table
owner.DSN_PREDICAT_TABLE.

Detailed cost table
owner.DSN_DETCOST_TABLE.

Sort table
owner.DSN_SORT_TABLE.

Sort key table
owner.DSN_SORTKEY_TABLE.

Filter table
owner.DSN_FILTER_TABLE.

Page range table
owner.DSN_PGRANGE_TABLE.

Parallel group table
owner.DSN_PGROUP_TABLE.

Parallel task table
owner.DSN_PTASK_TABLE.

View reference table
owner.DSN_VIEWREF_TABLE.

Query table
owner.DSN_QUERY_TABLE.

Query information table
owner.DSN_QUERYINFO_TABLE.

Predicate selectivity table
owner.DSN_PREDICATE_SELECTIVITY.

Statistics feedback table table
owner.DSN_STAT_FEEDBACK

Examples

Example 1: Determine the steps required to execute the query '`SELECT X.ACTNO...`'. Assume that no set of rows in the PLAN_TABLE has the value 13 for the QUERYNO column.

```
EXPLAIN PLAN SET QUERYNO = 13
FOR SELECT X.ACTNO, X.PROJNO, X.EMPNO, Y.JOB, Y.EDLEVEL
    FROM DSN8B10.EMPPROJACT X, DSN8B10.EMP Y
       WHERE X.EMPNO = Y.EMPNO
          AND X.EMPTIME > 0.5
          AND (Y.JOB = 'DESIGNER' OR Y.EDLEVEL >= 12)
       ORDER BY X.ACTNO, X.PROJNO;
```

Example 2: Retrieve the information returned in Example 1. Assume that a statement table exists, so also retrieve the estimated cost of processing the query. Use the following query, which joins the plan table and the statement table.

```
SELECT * FROM PLAN_TABLE A, DSN_STATEMNT_TABLE B
  WHERE A.QUERYNO = 13 and B.QUERYNO = 13
  ORDER BY A.QBLOCKNO, A.PLANNO, A.MIXOPSEQ;
```

Example 3: Have the cached statement with statement ID 124 explained. Assume that host variable SID contains 124.

```
EXPLAIN STMTCACHE STMTID :SID;
```

Example 4: Have one row of data for each statement in the dynamic statement cache written to the DSN_STATEMENT_CACHE_TABLE.

```
EXPLAIN STMTCACHE ALL;
```

Example 5: Assume that you want to use the plan table that was created by ADMF001 and your authorization ID is SYSADM. If you have an alias on ADMF001.PLAN_TABLE (CREATE ALIAS SYSADM.PLAN_TABLE FOR ADMF001.PLAN_TABLE) and sufficient INSERT and SELECT privileges on the table, the following EXPLAIN statement will execute and ADMF001.PLAN_TABLE will be populated.

```
EXPLAIN PLAN SET QUERYNO = 101
  FOR SELECT * FROM DSN8B10.EMP;
```

Example 6: Add explain information to the current user's PLAN_TABLE for all static SQL statements in the current copy of the package 'COLLA.PACK52604':

```
EXPLAIN PACKAGE COLLECTION 'COLLA' PACKAGE 'PACK52604' COPY 'CURRENT';
```

PSPI

Related concepts:

Interpreting data access by using EXPLAIN (DB2 Performance)

Related tasks:

Checking how DB2 resolves functions by using DSN_FUNCTION_TABLE (DB2 Application programming and SQL)

Capturing EXPLAIN information (DB2 Performance)

Related reference:

EXPLAIN bind option (DB2 Commands)

EXPLAIN tables (DB2 Performance)

FETCH

The FETCH statement positions a cursor on a row of its result table. It can return zero, one, or multiple rows and assigns the values of the rows to variables if there is a target specification.

There are two forms of this statement:

- **Single row fetch:** positions the cursor and, optionally, retrieves data from a single row of the result table.
- **Multiple row fetch:** positions the cursor on zero or more rows of the result table and, optionally, returns data if there is a target specification.

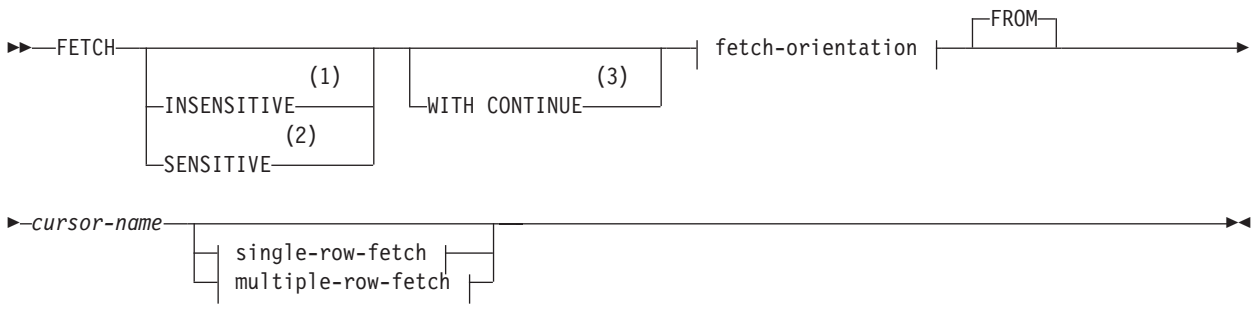
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. Multiple row fetch is not supported in REXX, Fortran, or SQL Procedure applications[35]. The FETCH statement with the WITH CONTINUE clause is not supported in REXX.

Authorization

See "DECLARE CURSOR" on page 1535 for an explanation of the authorization required to use a cursor.

Syntax



Notes:

1 The default depends on the sensitivity of the cursor. If INSENSITIVE is specified on the DECLARE CURSOR, then the default is INSENSITIVE and if SENSITIVE is specified on the DECLARE CURSOR, then the default is SENSITIVE.

2 If INSENSITIVE or SENSITIVE is specified, *single-row-fetch* or *multiple-row-fetch* must be specified.

3 If WITH CONTINUE is specified, *single-row-fetch* must be specified.

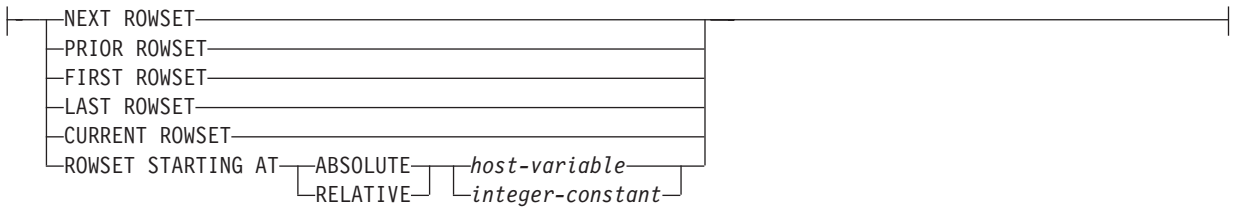
fetch-orientation

35. ASSEMBLER and other languages are supported, but this support is limited to statements that allow USING DESCRIPTOR. The precompiler does not recognize host-variable-arrays except in C/C++, COBOL, and PL/I.

fetch-orientation:

row-positioned:

rowset-positioned:

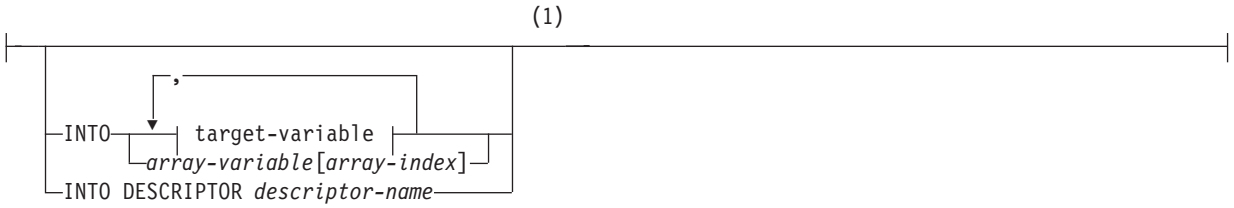


Notes:

1 If BEFORE or AFTER is specified, SENSITIVE, INSENSITIVE, single-row-fetch, or multiple-row-fetch must not be specified.

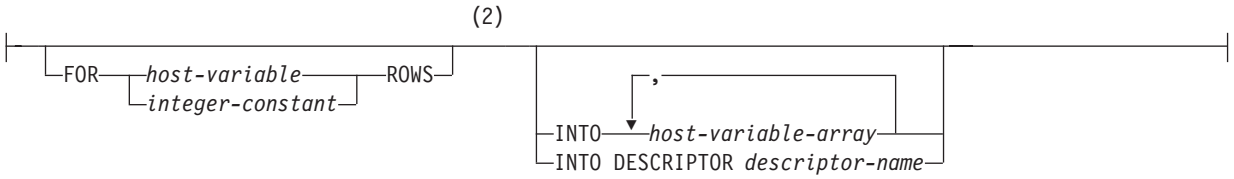
fetch-type

single-row-fetch:



target-variable:

multiple-row-fetch:



Notes:

1 For single-row-fetch, a host-variable-array can be specified instead of a host variable and the descriptor can describe host-variable-arrays. In either case, data is returned only for the first entry of the host-variable-array.

2 This clause is optional. If this clause is not specified and either a rowset size has not been established yet or a row positioned FETCH statement was the last type of FETCH statement issued for this cursor, the rowset size is implicitly one. If the last FETCH statement issued for this cursor was a rowset positioned FETCH statement and this clause is not specified, the rowset size is the same size as the previous rowset positioned FETCH.

Description

INSENSITIVE

Returns the row from the result table as it is. If the row has been previously fetched with a FETCH SENSITIVE, it reflects changes made outside this cursor before the FETCH SENSITIVE statement was issued. Positioned updates and deletes are reflected with FETCH INSENSITIVE if the same cursor was used for the positioned update or delete.

INSENSITIVE can only be specified for cursors declared as INSENSITIVE or SENSITIVE STATIC (or if the cursor is declared as ASENSITIVE and DB2 defaults to INSENSITIVE). Otherwise, if the cursor is declared as SENSITIVE DYNAMIC (or if the cursor is declared as ASENSITIVE and DB2 defaults to SENSITIVE DYNAMIC), an error occurs and the FETCH statement has no effect. For an INSENSITIVE cursor, specifying INSENSITIVE is optional because it is the default.

SENSITIVE

Updates the fetched row in the result table from the corresponding row in the base table of the cursor's SELECT statement and returns the current values. Thus, it reflects changes made outside this cursor. SENSITIVE can only be specified for a sensitive cursor. Otherwise, if the cursor is insensitive, an error occurs and the FETCH statement has no effect. For a SENSITIVE cursor, specifying SENSITIVE is optional because it is the default.

When the cursor is declared as SENSITIVE STATIC and a FETCH SENSITIVE is requested, the following steps are taken:

1. DB2 retrieves the row of the database that corresponds to the row of the result table that is about to be fetched.
2. If the corresponding row has been deleted, a "delete hole" occurs in the result table, a warning is issued, the cursor is repositioned on the "hole", and no data is fetched. (DB2 marks a row in the result table as a "delete hole" when the corresponding row in the database is deleted.)
3. If the corresponding row has not been deleted, the predicate of the underlying SELECT statement is re-evaluated. If the row no longer satisfies the predicate, an "update hole" occurs in the result table, a warning is issued, the cursor is repositioned on the "hole," and no data is fetched. (DB2 marks a row in the result table as an "update hole" when an update to the corresponding row in the database causes the row to no longer qualify for the result table.)
4. If the corresponding row does not result in a delete or an update hole in the result table, the cursor is repositioned on the row of the result table and the data is fetched.

WITH CONTINUE

Specifies that the DB2 subsystem should prepare to allow subsequent FETCH CURRENT CONTINUE operations to access any truncated LOB or XML result column following an initial FETCH operation that provides output variables that are not large enough to hold the entire LOB or XML columns. When the WITH CONTINUE clause is specified, the DB2 subsystem takes the following actions that can differ from the case where the FETCH statement does not include the WITH CONTINUE clause:

- If truncation occurs when returning an XML or LOB column, the DB2 subsystem will remember the truncation position and will not discard the remaining data.
- If truncation occurs when returning an XML or LOB column, the DB2 subsystem returns the total length that would have been required to hold all of the data of the LOB or XML column. This will either be in the first four bytes of the LOB host variable structure or in the 4 byte area that is pointed to by the SQLDATALEN pointer in the SQLVAR entry of the SQLDA for that host variable. What is returned depends on the programming method that is used. See “SQL descriptor area (SQLDA)” on page 2079 for details about the SQLDA contents.
- If returning XML data, the result column will be fully materialized in the database before the data is returned.

If the CURRENT CONTINUE clause is specified, the WITH CONTINUE behavior is assumed.

AFTER

Positions the cursor after the last row of the result table. Values are not assigned to host variables. The number of rows of the result table are returned

in the SQLERRD1 and SQLERRD2 fields of the SQLCA for cursors with an effective sensitivity of INSENSITIVE or SENSITIVE STATIC.

BEFORE
Positions the cursor before the first row of the result table. Values are not assigned to host variables.

row-positioned
Positioning of the cursor with row-positioned fetch orientations NEXT, PRIOR, CURRENT and RELATIVE is done in relation to the current cursor position. Following a successful row-positioned FETCH statement, the cursor is positioned on a single row of data. If the cursor is enabled for rowsets, positioning is performed relative to the current row or the first row of the current rowset, and the cursor is positioned on a rowset consisting of a single row.

NEXT
Positions the cursor on the next row or rows of the result table relative to the current cursor position, and returns data if a target is specified. NEXT is the only row-positioned fetch operation that can be explicitly specified for cursors that are defined as NO SCROLL. NEXT is the default if no other cursor positioning is specified. If a specified row reflects a hole, a warning is issued and data values are not assigned to host variables for that row.

Table 139 lists situations for different cursor positions and the results when NEXT is used.

Table 139. Results when NEXT is used with different cursor positions

Current state of the cursor	Result of FETCH NEXT
Before the first row	Cursor is positioned on the first row[1] and data is returned if requested.
On the last row or after the last row	A warning occurs, values are not assigned to host variables, and the cursor position is unchanged.
Before a hole	For a SENSITIVE STATIC cursor, a warning occurs for a delete hole or an update hole, values are not assigned to host variables, and the cursor is positioned on the hole.
Unknown	An error occurs, values are not assigned to host variables, and the cursor position remains unknown.

Note:

1. This row is not applicable in the case of a forward-only cursor (that is when NO SCROLL was specified implicitly or explicitly).

PRIOR
Positions the cursor on the previous row or rows of the result table relative to the current cursor position, and returns data if a target is specified. If a specified row reflects a hole, a warning is issued, and data values are not assigned to host variables for that row.

Table 140 on page 1655 lists situations for different cursor positions and the results when PRIOR is used.

Table 140. Results when PRIOR is used with different cursor positions

Current state of the cursor	Result of FETCH PRIOR
Before the first row or on the first row	A warning occurs, values are not assigned to host variables, and the cursor position is unchanged.
After a hole	For a SENSITIVE STATIC cursor, a warning occurs for a delete hole or an update hole, values are not assigned to host variables, and the cursor is positioned on the hole.
After the last row	Cursor is positioned on the last row.
Unknown	An error occurs, values are not assigned to host variables, and the cursor position remains unknown.

FIRST
Positions the cursor on the first row of the result table, and returns data if a target is specified. For a SENSITIVE STATIC cursor, if the first row of the result table is a hole, a warning occurs for a delete hole or an update hole and values are not assigned to host variables.

LAST
Positions the cursor on the last row of the result table, and returns data if a target is specified. The number of rows of the result table is returned in the SQLERRD1 and SQLERRD2 fields of the SQLCA for an insensitive or sensitive static cursor. For a SENSITIVE STATIC cursor, if the last row of the result table is a hole, a warning occurs for a delete hole or an update hole and values are not assigned to host variables.

CURRENT
The cursor position is not changed, data is returned if a target is specified. If the cursor was positioned on a rowset of more than one row, the cursor position is on the first row of the rowset.

Table 141 lists situations in which errors occur with the CURRENT clause.

Table 141. Situations in which errors occur with CURRENT

Current state of the cursor	Result of FETCH CURRENT
Before the first row or after the last row	A warning occurs, values are not assigned to host variables, and the cursor position is unchanged.
On a hole	For a SENSITIVE STATIC, a warning occurs for a delete hole or an update hole, values are not assigned to host variables, and the cursor is positioned on the hole. If the cursor is defined as a rowset cursor, with isolation level UR or a sensitive dynamic scrollable cursor, it is possible that a different row will be returned than the FETCH that established the most recent cursor position. This can occur while fetching a row again when it is determined to not be there anymore. In this case, fetching continues moving forward to get the row of data.

Table 141. Situations in which errors occur with CURRENT (continued)

Current state of the cursor	Result of FETCH CURRENT
Unknown	An error occurs, values are not assigned to host variables, and the cursor position remains unknown.

CONTINUE

The cursor positioning is not changed, and data is returned if a target is specified. The FETCH CURRENT CONTINUE statement retrieves remaining data for any LOB or XML column result values that were truncated on a previous FETCH or FETCH CURRENT CONTINUE statement. It assigns the remaining data for those truncated columns to the host variables that are referenced in the statement or pointed to by the descriptor. The data that is returned for previously-truncated result values begins at the point of truncation. This form of the CURRENT clause must only be used after a single-row FETCH WITH CONTINUE or FETCH CURRENT CONTINUE statement that has returned partial data for one or more LOB or XML columns. The cursor must be open and positioned on a row.

FETCH CURRENT CONTINUE must pass host variables entries for all columns in the SELECT list, even though the non-LOB columns or non-XML columns will not return any data.

ABSOLUTE

host-variable or *integer-constant* is assigned to an integral value *k*. If a *host-variable* is specified, it must be an exact numeric type with zero scale and must not include an indicator variable. The possible data types for the host variable are DECIMAL(*n*,0) or integer. The DECIMAL data type is limited to DECIMAL(18,0). An *integer-constant* can be up to 31 digits, depending on the application language.

If *k*=0, the cursor is positioned before the first row of the result table. Otherwise, ABSOLUTE positions the cursor to row *k* of the result table if *k*>0, or to *k* rows from the bottom of the table if *k*<0. For example, "ABSOLUTE -1" is the same as "LAST".

Data is returned if the specified position is within the rows of the result table, and a target is specified.

If an absolute position is specified that is before the first row or after the last row of the result table, a warning occurs, values are not assigned to host variables, and the cursor is positioned either before the first row or after the last row. If the resulting cursor position is after the last row for INSENSITIVE and SENSITIVE STATIC scrollable cursors, the number of rows of the result table are returned in the SQLERRD1 and SQLERRD2 fields of the SQLCA. If row *k* of the result table is a hole, a warning occurs and values are not assigned to host variables.

FETCH ABSOLUTE 0 results in positioning before the first row and a warning is issued. FETCH BEFORE results in positioning before the first row and no warning is issued.

Table 142 lists some synonymous specifications.

Table 142. Synonymous scroll specifications for ABSOLUTE

Specification	Alternative
ABSOLUTE 0 (but with a warning)	BEFORE (without a warning)

Table 142. Synonymous scroll specifications for ABSOLUTE (continued)

Specification	Alternative
ABSOLUTE +1	FIRST
ABSOLUTE -1	LAST
ABSOLUTE -*m*, 0<m≤*n*	ABSOLUTE *n*+1-*m*
ABSOLUTE *n*	LAST
ABSOLUTE -*n*	FIRST
ABSOLUTE *x* (with a warning)	AFTER (without a warning)
ABSOLUTE -*x* (with a warning)	BEFORE (without a warning)

Note: Assume: 0<=*m*<=*n*<*x* Where, *n* is the number of rows in the result table.

RELATIVE

host-variable or *integer-constant* is assigned to an integral value *k*. If a *host-variable* is specified, it must be an exact numeric type with zero scale and must not include an indicator variable. The possible data types for the host variable are DECIMAL(*n*,0) or integer. The DECIMAL data type is limited to DECIMAL(18,0).

If the cursor is positioned before the first row, or after the last row of the result table, the cursor position is determined as follows:

- If *n* is 0, the cursor position is unchanged, values are not assigned to host variables, and a warning occurs
- If *n* is positive, and the cursor is positioned before the first row, the cursor is positioned on a rowset starting at row *n*
- If *n* is positive, and the cursor is positioned after the last row, a warning occurs
- If *n* is negative, and the cursor is positioned before the first row, a warning occurs
- If *n* is negative, and the cursor is positioned after the last row, the cursor is positioned on a rowset starting as row *n* from the end of the result table

An *integer-constant* can be up to 31 digits, depending on the application language.

Data is returned if the specified position is within the rows of the result table, and a target is specified.

RELATIVE positions the cursor to the row in the result table that is either *k* rows after the current row if *k*>0, or ABS*(k)* rows before the current row if *k*<0. For example, "RELATIVE -1" is the same as "PRIOR". If *k*=0, the position of the cursor does not change (that is, "RELATIVE 0" is the same as "CURRENT").

If a relative position is specified that results in positioning before the first row or after the last row, a warning is issued, values are not assigned to host variables, and the cursor is positioned either before the first row or after the last row. If the resulting cursor position is after the last row for INSENSITIVE and SENSITIVE STATIC scrollable cursors, the number of rows of the result table is returned in the SQLERRD1 and SQLERRD2 fields of the SQLCA. If the cursor is positioned on a hole and RELATIVE 0 is specified or if the target row is a hole, a warning occurs and values are not assigned to host variables.

If the cursor is defined as a rowset cursor, with isolation level UR or a sensitive dynamic scrollable cursor, it is possible that a different row will be returned than the FETCH that established the most recent cursor position. This can occur while fetching a row again when it is determined to not be there anymore. In this case, fetching continues moving forward to get the row data.

If the cursor position is unknown and RELATIVE 0 is specified, an error occurs.

Table 143 lists some synonymous specifications.

Table 143. Synonymous Scroll Specifications for RELATIVE

Specification	Alternative
RELATIVE +1	NEXT
RELATIVE -1	PRIOR
RELATIVE 0	CURRENT
RELATIVE +r (with a warning)	AFTER (without a warning)
RELATIVE -r (with a warning)	BEFORE (without a warning)

Note:

r has to be large enough to position the cursor beyond either end of the result table.

rowset-positioned

Positioning of the cursor with rowset-positioned fetch orientations NEXT ROWSET, PRIOR ROWSET, CURRENT ROWSET, and ROWSET STARTING AT RELATIVE is done in relation to the current cursor position. Following a successful row-positioned FETCH statement, the cursor is positioned on a rowset of data. The number of rows in the rowset is determined either explicitly or implicitly. The FOR *n* ROWS clause in the multiple-row-fetch clause is used to explicitly specify the size of the rowset. Positioning is performed relative to the current row or first row of the current rowset, and the cursor is positioned on all rows of the rowset.

A rowset-positioned fetch orientation must not be specified if the current cursor position is not defined to access rowsets. NEXT ROWSET is the only rowset-positioned fetch orientation that can be specified for cursors that are defined as NO SCROLL.

If a row of the rowset reflects a hole, a warning is returned, data values are not assigned to host variable arrays for that row (that is, the corresponding positions in the target host variable arrays are untouched), and -3 is returned in all provided indicator variables for that row. If a hole is detected, and at least one indicator variable is not provided, an error occurs.

NEXT ROWSET

Positions the cursor on the next rowset of the result table relative to the current cursor position, and returns data if a target is specified. The next rowset is logically obtained by fetching the row that follows the current rowset and fetching additional rows until the number of rows that is specified implicitly or explicitly in the FOR n ROWS clause is obtained or the last row of the result table is reached.

If the cursor is positioned before the first row of the result table, the cursor is positioned on the first rowset.

If the cursor is positioned on the last row or after the last row of the result table, the cursor position is unchanged, values are not assigned to host variable arrays, and a warning occurs.

If a row of the rowset reflects a hole, the following actions occur:
- A warning is returned.
- Data values are not assigned to the host-variable-arrays for that row (that is, the corresponding positions in the target host-variable-arrays are untouched).
- A value of -3 is returned in all of the indicator variables that are provided for the row.

If a hole is detected and at least one indicator variable is not provided, an error is returned.

If the cursor is not positioned because of a prior error, values are not assigned to the host-variable-array, and an error is returned. If a row of the rowset would be after the last row of the result table, values are not assigned to host-variable-arrays for that row and any subsequent requested rows of the rowset, and a warning is returned.

NEXT ROWSET is the only rowset positioned fetch orientation that can be explicitly be specified for cursors that are defined as NO SCROLL.

PRIOR ROWSET

Positions the cursor on the previous rowset of the result table relative to the current position, and returns data if a target is specified.

The prior rowset is logically obtained by fetching the row that precedes the current rowset and fetching additional rows until the number of rows that is specified implicitly or explicitly in the FOR n ROWS clause is obtained or the last row of the result table is reached.

If the cursor is positioned after the last row of the result table, the cursor is positioned on the last rowset.

If the cursor is positioned before the first row or on the first row of the result table, the cursor position is unchanged, values are not assigned to host variable arrays, and a warning occurs.

If a row would be before the first row of the result table, the cursor is positioned on a partial rowset that consists of only those rows that are prior to the current position of the cursor starting with the first row of the result table, and a warning is returned. Values are not assigned to the host-variable-arrays for the rows in the rowset for which the warning is returned.

Although the rowset is logically obtained by fetching backwards from before the current rowset, the data is returned to the application starting with the first row of the rowset, to the end of the rowset.

If a row of the rowset reflects a hole, the following actions occur:
- A warning is returned.
- Data values are not assigned to the host-variable-arrays for that row (that is, the corresponding positions in the target host-variable-arrays are untouched).
- A value of -3 is returned in all of the indicator variables that are provided for the row.

If a hole is detected and at least one indicator variable is not provided, an error is returned.

If the cursor is not positioned because of a prior error, values are not assigned to the host-variable-array, and an error is returned.

FIRST ROWSET

Positions the cursor on the first rowset of the result table, and returns data if a target is specified.

If a row of the rowset reflects a hole, the following actions occur:

- A warning is returned.
- Data values are not assigned to the host-variable-arrays for that row (that is, the corresponding positions in the target host-variable-arrays are untouched).
- A value of -3 is returned in all of the indicator variables that are provided for the row.

If a hole is detected and at least one indicator variable is not provided, an error is returned.

If the result table contains fewer rows than specified implicitly or explicitly in the FOR *n* ROWS clause, values are not assigned to host-variable-arrays after the last row of the result table, and a warning is returned.

LAST ROWSET

Positions the cursor on the last rowset of the result table and returns data if a target is specified. The last rowset is logically obtained by fetching the last row of the result table and fetching prior rows until the number of rows in the rowset is obtained or the first row of the result table is reached. Although the rowset is logically obtained by fetching backwards from the bottom of the result table, the data is returned to the application starting with the first row of the rowset, to the end of the rowset, which is also the end of the result table.

If a row of the rowset reflects a hole, the following actions occur:

- A warning is returned.
- Data values are not assigned to the host-variable-arrays for that row (that is, the corresponding positions in the target host-variable-arrays are untouched).
- A value of -3 is returned in all of the indicator variables that are provided for the row.

If a hole is detected and at least one indicator variable is not provided, an error is returned.

If the result table contains fewer rows than specified implicitly or explicitly in the FOR *n* ROWS clause, the last rowset is the same as the first rowset, values are not assigned to host-variable-arrays after the last row of the result table, and a warning is returned.

CURRENT ROWSET

If the FOR *n* ROWS clause specifies a number different from the number of rows specified implicitly or explicitly in the FOR *n* ROWS clause on the most recent FETCH statement for this cursor, the cursor is repositioned on the specified number of rows, starting with the first row of the current rowset. If the cursor is positioned before the first row, or after the last row of the result table, the cursor position is unchanged, values are not assigned to host variable arrays, and a warning occurs. If the FOR *n* ROWS clause is not specified, it is possible that the FETCH statement will position the cursor on a partial rowset when the FETCH CURRENT ROWSET statement is processed. In this case, DB2 attempts to position the cursor on

a full rowset starting with the first row of the current rowset. Otherwise, the position of the cursor on the current rowset is unchanged. Data is returned if a target is specified.

With isolation level UR or a sensitive dynamic scrollable cursor, it is possible that different rows will be returned than the FETCH that established the most recent rowset cursor position. This can occur while refetching the first row of the rowset when it is determined to not be there anymore. In this case, fetching continues moving forward to get the first row of data for the rowset. This can also occur when changes have been made to other rows in the current rowset such that they no longer exist or have been logically moved within (or out of) the result table of the cursor.

If the cursor is not positioned because of a prior error, values are not assigned to the host-variable-array, and an error occurs.

If the current rowset contains fewer rows than specified implicitly or explicitly in the FOR *n* ROWS clause, values are not assigned to host-variable-arrays after the last row, and a warning is returned.

ROWSET STARTING AT ABSOLUTE or RELATIVE *host-variable* **or** *integer-constant*

Positions the cursor on the rowset beginning at the row of the result table that is indicated by the ABSOLUTE or RELATIVE specification, and returns data if a target is specified.

host-variable or *integer-constant* is assigned to an integral value *k*. If *host-variable* is specified, it must be an exact numeric type with scale zero, and must not include an indicator variable. The possible data types for the host variable are DECIMAL(n,0) or integer, where the DECIMAL data type is limited to DECIMAL(18,0). If a constant is specified, the value must be an integer.

If a row of the result table would be after the last row or before the first row of the result table, values are not assigned to host-variable-arrays for that row and a warning is returned.

ABSOLUTE

If k=0, an error occurs. If k>0, the first row of the rowset is row k. If k<0, the rowset is positioned on the ABS(k) rows from the bottom of the result table. Assume that ABS(k) is equal to the number of rows for the rowset and that there are enough row to return a complete rowset:

- FETCH ROWSET STARTING AT ABSOLUTE -k is the same as FETCH LAST ROWSET.
- FETCH ROWSET STARTING AT ABSOLUTE 1 is the same as FETCH FIRST ROWSET.

RELATIVE

If k=0 and the FOR *n* ROWS clause does not specify a number different from the number most recently specified implicitly or explicitly for this cursor, then the position of the cursor does not change (that is, "RELATIVE ROWSET 0" is the same as "CURRENT ROWSET"). If k=0 and the FOR *n* ROWS clause specifies a number different from the number most recently specified implicitly or explicitly for this cursor, then the cursor is repositioned on the specified number of rows, starting with the first row of the current rowset.

If the cursor is positioned before the first row, or after the last row of the result table, the cursor position is determined as follows:

- If n is 0, the cursor position is unchanged, values are not assigned to host variables, and a warning occurs. This is the same as FETCH CURRENT ROWSET.
- If *n* is positive, and the cursor is positioned before the first row, the cursor is positioned on a rowset starting a row *n*.
- If *n* is positive, and the cursor is positioned after the last row, a warning occurs.
- If *n* is negative, and the cursor is positioned before the first row, a warning occurs.
- If *n* is negative, and the cursor is positioned after the last row, the cursor is positioned on a rowset starting at row *n* from the bottom of the result table.

Otherwise, RELATIVE repositions the cursor so that the first row of the new rowset cursor position is on the row in the result table that is either k rows after the first row of the current rowset cursor position if k>0, or ABS(k) rows before the first row of the current rowset cursor position if k<0. Assume that ABS(k) is equal to the number of rows for the resulting rowset

- FETCH ROWSET STARTING AT RELATIVE -k is the same as FETCH PRIOR ROWSET.
- FETCH ROWSET STARTING AT RELATIVE k is the same as FETCH NEXT ROWSET.
- FETCH ROWSET STARTING AT RELATIVE 0 is the same as FETCH CURRENT ROWSET.

When ROWSET STARTING AT RELATIVE -n is specified and there are not enough rows between the current position of the cursor and the beginning of the result table to return a complete rowset:
- A warning is returned.
- Values are not assigned to the host-variable-arrays.
- The cursor is positioned before the first row.

If a row of the rowset reflects a hole, If a row of the rowset reflects a hole, the following actions occur:
- A warning is returned.
- Data values are not assigned to the host-variable-arrays for that row (that is, the corresponding positions in the target host-variable-arrays are untouched).
- A value of -3 is returned in all of the indicator variables that are provided for the row.

If a hole is detected and at least one indicator variable is not provided, an error is returned. If a row of the rowset is unknown, values are not assigned to host variable arrays for that row, and an error is returned. If a row of the rowset would be after the last row or before the first row of the result table, values are not assigned to host-variable-arrays for that row, and a warning is returned.

cursor-name

Identifies the cursor to be used in the fetch operation. The cursor name must identify a declared cursor, as explained in the description of the DECLARE CURSOR statement in "DECLARE CURSOR" on page 1535, or an allocated cursor, as explained in "ALLOCATE CURSOR" on page 847. When the FETCH statement is executed, the cursor must be in the open state.

If a single-row-fetch or multiple-row-fetch clause is not specified, the cursor position is adjusted as specified, but no data is returned to the user.

single-row-fetch
When *single-row-fetch* is specified, SENSITIVE or INSENSITIVE can be specified though there is a default. The default depends on the sensitivity of the cursor. If the sensitivity of the cursor is INSENSITIVE, then the default is INSENSITIVE. If the effective sensitivity of the cursor is SENSITIVE DYNAMIC or SENSITIVE STATIC, then the default is SENSITIVE. The single-row-fetch or multiple-row-fetch clause must not be specified when the FETCH BEFORE or FETCH AFTER option is specified. They are required when FETCH BEFORE or FETCH AFTER is not specified. If an individual fetch operation causes the cursor to be positioned or to remain positioned on a row if there is a target specification, the values of the result table are assigned to host variables as specified by the single-fetch-clause.

INTO *target-variable* **or** *array-variable*[*array-index*]
Identifies one or more targets for the assignment of output values. The number of targets in the INTO clause must equal the number of values that are to be assigned. The first value in the result row is assigned to the first target in the list, the second value to the second target, and so on. A target variable must not be specified more than once in the INTO clause. Each assignment to a target is made in sequence through the list according to the rules described in "Assignment and comparison" on page 121.

The value 'W' is assigned to the SQLWARN3 field of the SQLCA if the number of targets is less than the number of result column values.

If an error occurs on any assignment, the value is not assigned to the target, and no more values are assigned to the specified targets. Any values that have already been assigned remain assigned.

host-variable-name
Identifies the host variable that is the assignment target. For LOB output values, the target can be a regular host variable (if it is large enough), a LOB locator variable, or a LOB file reference variable.

SQL-parameter-name
Identifies the parameter that is the assignment target.

SQL-variable-name
Identifies the SQL variable that is the assignment target. SQL variables must be declared before they are used.

array-variable [*array-index*]
Specifies an array element that is the target of the assignment.

An array element must not be specified as the target for an assignment if *common-table-expression* is also specified in the statement.

[*array-index*]
An expression that specifies which element in the array is the target of the assignment.

For an ordinary array, the array index expression must be castable to INTEGER, and must not be the null value. The index value must be between 1 and the maximum cardinality that is defined for the array.

For an associative array, the array index expression must be castable to the index data type of the associative array, and must not be the null value.

array-index must not be:

- An expression that references the CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP special register
- A nondeterministic function
- A function that is defined with EXTERNAL ACTION
- A function that is defined with MODIFIES SQL DATA
- A sequence expression

INTO DESCRIPTOR *descriptor-name*
Identifies an SQLDA that contains a valid description of the host output variables. Result values from the associated SELECT statement are returned to the application program in the output host variables.

Before the FETCH statement is processed, you must set the following fields in the SQLDA:

- SQLN to indicate the number of SQLVAR occurrences provided in the SQLDA

  A REXX SQLDA does not contain this field.
- SQLABC to indicate the number of bytes of storage allocated in the SQLDA
- SQLD to indicate the number of variables used in the SQLDA when processing the statement
- SQLVAR occurrences to indicate the attributes of the variables

The SQLDA must have enough storage to contain all SQLVAR occurrences. Each SQLVAR occurrence describes a host variable or buffer into which a value in the result table is to be assigned. If LOBs are present in the results, there must be additional SQLVAR entries for each column of the result table. If the result table contains only base types and distinct types, multiple SQLVAR entries are not needed for each column. However, extra SQLVAR entries are needed for distinct types as well as for LOBs in DESCRIBE and PREPARE INTO statements. For more information on the SQLDA, which includes a description of the SQLVAR and an explanation on how to determine the number of SQLVAR occurrences, see "SQL descriptor area (SQLDA)" on page 2079.

SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN.

See "Identifying an SQLDA in C or C++" on page 2099 for how to represent *descriptor-name* in C.

multiple-row-fetch
Retrieves multiple rows of data from the result table of a query. The FOR *n* ROWS clause of the FETCH statement controls how many rows are returned on a single FETCH statement. The fetch orientation determines whether the resulting cursor position (for example, on a single row, rowset, before, or after the result table). Fetching stops when an error is returned, all requested rows are fetched, or the end of data condition is reached.

Fetching multiple rows of data can be done with scrollable or non-scrollable cursors. The operations used to define, open, and close a cursor used for fetching multiple rows of data are the same as for those used for single row FETCH statements.

If the BEFORE or AFTER option is specified, neither single-row-fetch or multiple-row-fetch can be specified.

FOR *host-variable* **or** *integer-constant* **ROWS**
host-variable or *integer-constant* is assigned to an integral value *k*. If a host variable is specified, it must be an exact numeric type with a scale of zero and must not include an indicator variable. Furthermore, *k* must be in the range, 0<k<=32767.

This clause must not be specified if a row-positioned fetch-orientation clause was specified. This clause must also not be specified for a cursor that is defined without rowset access.

If a rowset fetch orientation is specified and this clause is not specified, the number of rows in the resulting rowset is determined as follows:

- If the most recent FETCH statement for this cursor was a rowset-positioned FETCH, the number of rows of the rowset is implicitly determined by the number of rows that was most recently specified (implicitly or explicitly) for this cursor.
- When the most recent FETCH statement for this cursor was either FETCH BEFORE or FETCH AFTER and the most recent FETCH statement for this cursor prior to that was a rowset-positioned FETCH, the number of rows of the rowset is implicitly determined by the number of rows that were most recently specified (implicitly or explicitly) for this cursor.
- Otherwise, the rowset consists of a single row.

For result set cursors, the number of rows for a rowset cursor position, established in the procedure that defined the rowset, is not inherited by the caller when the rowset is returned. Use the FOR *n* ROWS clause on the first rowset FETCH statement for the result set in the calling program to establish the number of rows for the cursor. Otherwise, the rowset consists of a single row.

The cursor is positioned on the row or rowset that is specified by the orientation clause (for example, NEXT ROWSET), and those rows are fetched if a target is specified. After the cursor is positioned on the first row being fetched, the next k-1 rows are fetched. Fetching moves forward from the cursor position in the result table and continues until the end of data condition is returned, k-1 rows have been fetched, or an assignment error is returned.

The resulting cursor position depends on the fetch orientation that is specified:

- For a row-positioned fetch orientation, the cursor is positioned at the last row successfully retrieved.
- For a rowset-positioned fetch orientation, the cursor is positioned on all the rows retrieved.

The values from each individual fetch are placed in data areas that are described in the INTO or USING clause. If a target specification is provided for a rowset-positioned FETCH, the host variable arrays must be specified as the target specification, and the arrays must be defined with a dimension of 1 or greater. The target specification must be defined as an

array for a rowset-positioned FETCH even if the number of rows that is specified implicitly or explicitly is one. See Diagnostics information for rowset positioned FETCH statements.

INTO *host-variable-array*
Identifies for each column of the result table a host-variable-array to receive the data that is retrieved with this FETCH statement. If the number of host-variable-arrays is less than the number of columns of the result table, the SQLWARN3 field of the SQLCA is set to 'W'. No warning is given if there are more host-variable-arrays than the number of columns in the result table.

Each host-variable-array must be defined in the application program in accordance with the rules for declaring an array. A host-variable-array is used to return the values for a column of the result table. The number of rows to be fetched must be less than or equal to the dimension of each of the host-variable-arrays.

An optional indicator array can be specified for a host-variable-array. It should be specified if the SQLTYPE of any SQLVAR occurrence indicates that the column of the result table is nullable. Additionally, if an operation may result in null values, such as an UPDATE operation that results in a hole, is performed in the application, an indicator array should be specified. Otherwise an error occurs if null values are encountered. The indicators are returned as small integers.

INTO DESCRIPTOR *descriptor-name*
Identifies an SQLDA that must contain a valid description of zero or more host-variable-arrays or buffers into which the values for a column of the result table are to be returned.

Before the FETCH statement is processed, you must set the following fields in the SQLDA:

- SQLN to indicate the number of SQLVAR occurrences provided in the SQLDA.
- SQLABC to indicate the number of bytes of storage allocated for the SQLDA.
- SQLD to indicate the number of variables used in the SQLDA when processing the statement.
- SQLVAR occurrences to indicate the attributes of an element of the host-variable-array. Within each SQLVAR representing an array:
  - SQLTYPE indicates the data type of the elements of the host-variable-array.
  - SQLDATA field points to the first element of the host-variable-array.
  - The length fields (SQLLEN and SQLLONGLEN) are set to indicate the maximum length of a single element of the array.
  - SQLNAME - The length of SQLNAME must be set to 8, and the first two bytes of the data portion of SQLNAME must be initialized to X'0000'. The fifth and sixth bytes must contain a flag field and the seventh and eighth bytes must be initialized to a binary small integer (half word) representation of the dimension of the host-variable-array, and the corresponding indicator array, if one is specified.

  The SQLVAR entry for the number of rows must also contain a flag value. The number of rows to be fetched must be less than or equal to the dimension of each of the host variable arrays.

You set the SQLDATA and SQLIND pointers to the beginning of the corresponding arrays. The SQLDA must have enough storage to contain all SQLVAR occurrences. Each SQLVAR occurrence describes a host-variable-array or buffer into which the values for a column in the result table are to be returned. If any column of the result table is a LOB, two SQLVAR entries must be provided for each SQLVAR, and SQLN must be set to two times the number of SQLVARS. SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN.

Notes

Assignment to targets:

The *n*th target identified by the INTO clause or described in the SQLDA corresponds to the *n*th column of the result table of the cursor. The data type of target must be compatible with its corresponding value. If the value is numeric, the target must have the capacity to represent the whole part of the value. For a datetime value, the target must be a character string variable of a minimum length as defined in "String representations of datetime values" on page 101. When the target is a host variable, if the value is null, an indicator variable must be specified.

Assignments are made in sequence through the list. Each assignment to a target is made according to the rules described in Chapter 2, "Language elements," on page 53. If the number of targets is less than the number of values in the row, the SQLWARN3 field of the SQLCA is set to 'W'. There is no warning if there are more targets than the number of result columns. If the target is a host variable and the value is null, an indicator variable must be provided. If an assignment error occurs, the value is not assigned to the target and no more values are assigned to targets. Any values that have already been assigned to targets remain assigned.

If more than one assignment is included in the same assignment statement, all expressions are evaluated before the assignments are performed. For example, a reference to a variable in an expression always uses the value of the variable prior to any assignment in the assignment statement.

Normally, you use LOB locators to assign and retrieve data from LOB columns. However, because of compatibility rules, you can also use LOB locators to assign data to targets with other data types. For more information on using locators, see Saving storage when manipulating LOBs by using LOB locators (DB2 Application programming and SQL).

A timestamp without time zone value must not be assigned to a timestamp with time zone target.

The default encoding scheme for the data is the value in the bind option ENCODING, which is the option for application encoding. If this statement is used with functions such as LENGTH or SUBSTRING that are operating on LOB locators, and the LOB data that is specifies by the locator is in a different encoding scheme from the ENCODING bind option, LOB materialization and character conversion occur. To avoid LOB materialization and character conversion, select the LOB data from the SYSIBM.SYSDUMMYA, SYSIBM.SYSDUMMYE, or SYSIBM.SYSDUMMYU sample table.

Restrictions on using the WITH CONTINUE and CURRENT CONTINUE clauses:

When using the WITH CONTINUE clause, the DB2 system will only reserve truncated data for result set columns of the BLOB, CLOB,

DBCLOB, or XML data type, and only when the output host variable data type is the appropriate LOB data type.

If an application uses FETCH WITH CONTINUE, and truncated data remains after the FETCH operation, the application cannot perform any intervening operation on that cursor before performing the FETCH CURRENT CONTINUE. If intervening operations on that cursor are performed, the truncated data is lost.

FETCH CURRENT CONTINUE is not supported with multi-row fetch. Also, FETCH CURRENT CONTINUE is not supported for non-LOB and non-XML columns that have been truncated. If truncation occurs for these non-LOB and non-XML columns, the truncated data will be discarded as usual.

Result column evaluation considerations:

If an error occurs as the result of an arithmetic expression in the SELECT list of an outer SELECT statement (division by zero, or overflow) or a numeric conversion error occurs, the result is the null value. As in any other case of a null value, an indicator variable must be provided and the main variable is unchanged. In this case, however, the indicator variable is set to -2. Processing of the statement continues as if the error had not occurred. (However, this error causes a positive SQLCODE.) If you do not provide an indicator variable, a negative value is returned in the SQLCODE field of the SQLCA. Processing of the statement terminates when the error is encountered. No value is assigned to the host variable or to later variables, though any values that have already been assigned to variables remain assigned.

If the specified host variable is not large enough to contain the result, a warning is returned and W is assigned to SQLWARN1 in the SQLCA. The actual length of the result is returned in the indicator variable associated with the host-variable, if an indicator is provided. It is possible that a warning may not be returned on a FETCH operation. This occurs as a result of optimizations, such as the use of system temporary tables or blocking. It is also possible that the returned warning applies to a previously fetched row. When a datetime value is returned, the length of the variable must be large enough to store the complete value. Otherwise, a warning or an error is returned.

Cursor positioning:

An open cursor has three possible positions:

- Before a row
- On a row or rowset
- After the last row

When a scrollable or non-scrollable cursor is opened, it is positioned before the first row in the result table. If a cursor is on a row, that row is called the current row of the cursor. If a cursor is on a rowset, the rows are called the current rowset of the cursor.

A cursor referred to in an UPDATE or DELETE statement must be positioned on a row or rowset. A cursor can only be on a row or rowset as a result of a FETCH statement.

If the cursor was declared SENSITIVE STATIC SCROLL, a row may be a *hole*, from which no values may be fetched, updated, or deleted. Holes do not exist with sensitive dynamic cursors because there is no temporary

result table. For information about holes in the result table of a cursor, see *DB2 Application Programming and SQL Guide*.

For scrollable cursors, the cursor position after an error varies depending on the type of error:

- When an operation is attempted against an update or delete hole, or when an update or delete hole is detected, the cursor is positioned on the hole.
- When a FETCH operation is attempted past the end of file, the cursor is positioned after the last row.
- When a FETCH operation is attempted before the beginning of file, the cursor is positioned before the first row.
- When an error causes the cursor position to be invalid such as when a single row positioned update or positioned delete error occurs that causes a rollback, the cursor is closed.

Cursor position after exception condition:
If an error occurs during the execution of a fetch operation, the position of the cursor and the result of any later fetch is unpredictable. It is possible for an error to occur that makes the position of the cursor invalid, in which case the cursor is closed.

If an individual fetch operation specifies a destination that is outside the range of the cursor, a warning is issued (except for FETCH BEFORE or FETCH AFTER), the cursor is positioned before or after the result table, and values are not assigned to host variables.

Concurrency and scrollability:
The current row of a cursor cannot be updated or deleted by another application process if it is locked. Unless it is already locked because it was inserted or updated by the application process during the current unit of work, the current row of a cursor is not locked if:

- The isolation level is UR, or
- The isolation level is CS, and
  - The result table of the cursor is read-only
  - The bind option CURRENTDATA(NO) is in effect

A dynamic scrollable cursor is useful when it is more important to the application to see updated rows and newly inserted rows and there is no need to see deleted rows. The isolation level of CS should be used for maximum concurrency with dynamic scrollable cursors. Specifying an isolation level of RR or RS severely restricts the update of the table, thus defeating the purpose of a SENSITIVE DYNAMIC scrollable cursor. If the application needs a constant result table, a SENSITIVE STATIC scrollable cursor with an isolation level of CS should be used.

Sensitivity of SENSITIVE STATIC SCROLL cursors to database changes:
When SENSITIVE STATIC SCROLL has been declared, the following rules apply:

- For the result of an update operation to be visible within a cursor after "open," the update operation must be a positioned update executed against the cursor, or a FETCH SENSITIVE in a STATIC cursor must be executed against a row which has been updated by some other means (that is, a searched update, committed updates of others, or an update with another cursor in the same process).

- Another process can update the base table of the SELECT statement so that the current values no longer satisfy the WHERE clause. In this case, an "update hole" effectively exists during the time the values in the base table do not satisfy the WHERE clause, and the row is no longer accessible through the cursor. When an attempt is made to fetch a row that has been identified as an update hole, no values are returned, and a warning is issued.

  Under SENSITIVE STATIC SCROLL cursors, update holes are only identified during positioned update, positioned delete, and FETCH SENSITIVE operations. Each positioned update, positioned delete, and FETCH SENSITIVE operation does the necessary tests to determine if an update hole exists.
- For the result of a delete operation to be visible within a SENSITIVE STATIC SCROLL cursor, the delete operation must be a positioned delete executed against the cursor or a FETCH SENSITIVE in a STATIC cursor must be executed against a row that has been deleted by some other means (that is, a searched delete, committed deletes of others, or a delete with another cursor in the same process).
- Another process, or the even the same process, may delete a row in the base table of the SELECT statement so that a row of the cursor no longer has a corresponding row in the base table. In this case, a "delete hole" effectively exists, and that row is no longer accessible through the cursor. When an attempt is made to fetch a row that has been identified as a delete hole, no values are returned, and a warning is issued.

  Under SENSITIVE STATIC SCROLL cursors, delete holes are identified during positioned update, positioned delete, and FETCH SENSITIVE operations.
- Inserts into the base table or tables of SENSITIVE STATIC SCROLL cursors are not seen after the cursor is opened.

LOB locators:

When information is retrieved into LOB locators and it is not necessary to retain the locator across FETCH statements, it is a good practice to issue a FREE LOCATOR statement before issuing another FETCH statement because locator resources are limited.

Isolation level considerations:

The isolation level of the statement (specified implicitly or explicitly) can affect the result of a rowset-positioned FETCH statement. This is possible when changes are made to the tables underlying the cursor when isolation level UR is used with a dynamic scrollable cursor, or with other isolation levels when rows have been added by the application fetching from the cursor. These situations can occur with the following fetch orientations:

PRIOR ROWSET

With a dynamic scrollable cursor and isolation level UR, the content of a prior rowset can be affected by other activity within the table. It is possible that a row that previously qualified for the cursor, and was included as a member of the "prior" rowset, has since been deleted or modified before it is actually returned as part of the rowset for the current statement. To avoid this behavior, use an isolation level other than UR.

CURRENT ROWSET

With a dynamic scrollable cursor, additional rows can be added between rows that form the rowset that was returned to the user. With isolation level RR, these rows can only be added by the application

fetching from the cursor. For isolation levels other than RR, other applications can insert rows that can affect the results of a subsequent FETCH CURRENT ROWSET. To avoid this behavior, use a static scrollable cursor instead of a dynamic scrollable cursor.

LAST ROWSET
With a dynamic scrollable cursor and isolation level UR, the content of the last rowset can be affected by other activity within the table. It is possible that a row that previously qualified for the cursor, and was included as a member of the "last" rowset, has since been deleted or modified before it is actually returned as part of the rowset for the current statement. To avoid this behavior, use an isolation level other than UR.

ROWSET STARTING AT RELATIVE *-n* (where *-n* is a negative number)
With a dynamic scrollable cursor and isolation level UR, the content of a prior rowset can be affected by other activity within the table. It is possible that a row that previously qualified for the cursor, and was included as a member of the "prior" rowset, has since been deleted or modified before it is actually returned as part of the rowset for the current statement. To avoid this behavior, use an isolation level other than UR.

Row positioned and rowset positioned FETCH statement interaction:
The following table demonstrates the interaction between row positioned and rowset positioned FETCH statements. The table is based on the following assumptions:

- TABLE T1 has 15 rows
- CURSOR CS1 is declared as follows:

```
DECLARE CS1 SCROLL CURSOR WITH ROWSET POSITIONING FOR
SELECT * FROM T1;
```

- An OPEN CURSOR statement has been successfully executed for CURSOR CS1 and the FETCH statements in the table are executed in the order that they appear in the table.

Table 144. Interaction between row positioned and rowset positioned FETCH statements

FETCH Statement	Cursor Position
FETCH FIRST	Cursor is positioned on row 1.
FETCH FIRST ROWSET	Cursor is positioned on a rowset of size 1, consisting of row 1.
FETCH FIRST ROWSET FOR 5 ROWS	Cursor is positioned on a rowset of size 5, consisting of rows 1, 2, 3, 4, and 5.
FETCH CURRENT ROWSET	Cursor is positioned on a rowset of size 5, consisting of rows 1, 2, 3, 4, and 5.
FETCH CURRENT	Cursor is positioned on row 1
FETCH FIRST ROWSET FOR 5 ROWS	Cursor is positioned on a rowset of size 5, consisting of rows 1, 2, 3, 4, and 5.
FETCH **or** FETCH NEXT	Cursor is positioned on row 2.
FETCH NEXT ROWSET	Cursor is positioned on a rowset of size 1, consisting of row 3.
FETCH NEXT ROWSET FOR 3 ROWS	Cursor is positioned on a rowset of size 3, consisting of rows 4,5, and 6.
FETCH NEXT ROWSET	Cursor is positioned on a rowset of size 3, consisting of rows 7,8, and 9.

Table 144. Interaction between row positioned and rowset positioned FETCH statements (continued)

FETCH Statement	Cursor Position
FETCH LAST	Cursor is positioned on row 15.
FETCH LAST ROWSET FOR 2 ROWS	Cursor is positioned on a rowset of size 2, consisting of rows 14 and 15.
FETCH PRIOR ROWSET	Cursor is positioned on a rowset of size 2, consisting of rows 12 and 13.
FETCH ABSOLUTE 2	Cursor is positioned on row 2.
FETCH ROWSET STARTING AT ABSOLUTE 2 FOR 3 ROWS	Cursor is positioned on a rowset of size 3, consisting of rows 2, 3, and 4.
FETCH RELATIVE 2	Cursor is positioned on row 4.
FETCH ROWSET STARTING AT ABSOLUTE 2 FOR 4 ROWS	Cursor is positioned on a rowset of size 4, consisting of rows 2, 3, 4, and 5.
FETCH RELATIVE -1	Cursor is positioned on row 1.
FETCH ROWSET STARTING AT ABSOLUTE 3 FOR 2 ROWS	Cursor is positioned on a rowset of size 2, consisting of rows 3 and 4.
FETCH ROWSET STARTING AT RELATIVE 4	Cursor is positioned on a rowset of size 2, consisting of rows 7 and 8.
FETCH PRIOR	Cursor is positioned on row 6.
FETCH ROWSET STARTING AT ABSOLUTE 13 FOR 5 ROWS	Cursor is positioned on a rowset of size 3, consisting of rows 13, 14, and 15.
FETCH FIRST ROWSET	Cursor is positioned on a rowset of size 5, consisting of rows 1, 2, 3, 4, and 5. **Note:** Even though the previous FETCH statement returned only 3 rows because EOF was encountered, DB2 will remember that 5 rows were requested by the previous FETCH statement.

Considerations for using the FOR n ROWS clause with the FETCH FIRST n ROWS ONLY clause:

A clause specifying the number of rows that you want can be specified in the SELECT statement of a cursor, the FETCH statement for a cursor, or both. However, these clauses have different effects:

- In the SELECT statement, a FETCH FIRST *n* ROWS ONLY clause controls the maximum number of rows that can be accessed with the cursor. When a FETCH statement attempts to retrieve a row beyond the number specified in the FETCH FIRST n ROWS ONLY clause of the SELECT statement, an end-of-data condition occurs.
- In a FETCH statement, a FOR *n* ROWS clause controls the number of rows that are returned for a single FETCH statement.

Both of these clauses can be specified.

Diagnostics information for rowset positioned FETCH statements:

A single FETCH statement from a rowset cursor might encounter zero, one, or more conditions. If the current cursor position is not valid for the fetch orientation, a warning occurs and the statement terminates. If a warning or non-terminating error (such as a bind out error) occurs during the fetch of a row, processing continues. In this case, a summary message is returned for the FETCH statement, and additional information about each fetched row is available with the GET DIAGNOSTICS statement. Use the GET

DIAGNOSTICS statement to obtain information about all of the conditions that are encountered for one of these FETCH statements. See "GET DIAGNOSTICS" on page 1679 for more information.

The SQLCA returns some information about errors and warnings that are found while fetching from a rowset cursor. Processing stops when the end of data is encountered, or when a terminating condition occurs. After each FETCH statement from a rowset cursor, information is returned to the program through the SQLCA. The SQLCA is set as follows:

- SQLCODE contains the SQLCODE.
- SQLSTATE contains the SQLSTATE.
- SQLERRD1 and SQLERRD2 contain the number of rows of the result table if the cursor is positioned on the last row of the result table.
- SQLERRD3 contains the actual number of rows returned. If SQLERRD3 is less than the number of rows requested, an error or end-of-data condition occurred.
- SQLWARN flags are set to represent all the warnings that were accumulated while processing the FETCH statement.

Consider the following examples, where 10 rows are fetched with a single FETCH statement.

- **Example 1:** Assume that an error is detected on the 5th row. SQLERRD3 is set to 4 for the 4 returned rows, SQLSTATE is set to 22537, and SQLCODE is set to -354. This information is also available from the GET DIAGNOSTICS statement (the information that is returned is generated from connected server, which may differ across different servers). For example:

```
GET DIAGNOSTICS :num_rows = ROW_COUNT, :num_cond = NUMBER;
-- Results of the statement:
-- num_rows = 4 and num_cond = 1 (1 condition)

GET DIAGNOSTICS CONDITION 1 :sqlstate = RETURNED_SQLSTATE,
:sqlcode = DB2_RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;
-- Results of the statement:
-- sqlstate = 22537, sqlcode = -354, and row_num = 5
```

- **Example 2:** Assume that an end-of-data condition is detected on the 6th row and that the cursor does not have immediate sensitivity to updates. SQLERRD3 is set to 5 for the 5 returned rows, SQLSTATE is set to 02000, and SQLCODE is set to +100. This information is also available from the GET DIAGNOSTICS statement. For example:

```
GET DIAGNOSTICS :num_rows = ROW_COUNT, :num_cond = NUMBER;
-- Results of the statement:
-- num_rows = 5 and num_cond = 1 (1 condition)

GET DIAGNOSTICS CONDITION 1 :sqlstate = RETURNED_SQLSTATE,
:sqlcode = DB2_RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;
-- Results of the statement:
-- sqlstate = 02000, sqlcode = 100, and row_num = 6
```

- **Example 3:** Assume that an bind out error condition is detected on the 5th row, the condition is recorded, and processing continues. Also, assume that an end-of-data condition is detected on the 8th row. SQLERRD3 is set to 7 for the 7 returned rows, SQLSTATE is set to 02000, and SQLCODE is set to +100. Processing to complete the FETCH statement is performed, and the bind out error that occurred is noted. An additional SQLCODE is recorded for the bind out error. SQLCODE is set to –354, and SQLSTATE is set to 01668. Use the GET DIAGNOSTICS statement to determine what went on. For example:

```
GET DIAGNOSTICS :num_rows = ROW_COUNT, :num_cond = NUMBER;
-- Results of the statement:
-- num_rows = 7 and num_cond = 3 (3 conditions)
GET DIAGNOSTICS CONDITION 1 :sqlstate = RETURNED_SQLSTATE,
:sqlcode = RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;
-- Results of the statement:
-- sqlstate = 01668, sqlcode = -354, and row_num = 0
GET DIAGNOSTICS CONDITION 2 :sqlstate = RETURNED_SQLSTATE,
:sqlcode = RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;
-- Results of the statement:
-- sqlstate = 02000, sqlcode = 100, and row_num = 0
GET DIAGNOSTICS CONDITION 3 :sqlstate = RETURNED_SQLSTATE,
:sqlcode = RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;
-- Results of the statement:
-- sqlstate = 22003, sqlcode = -302, and row_num = 5
```

In some cases, DB2 returns a warning if indicator variables are provided, or an error if indicator variables are not provided. These errors can be thought of as data mapping errors that result in a warning if indicator variables are provided.

- If indicator variables are provided, DB2 returns all rows to the user, marking the errors in the indicator variables. The SQLCODE and SQLSTATE contain the warning from the last data mapping error. The GET DIAGNOSTICS statement can be used to retrieve information about all the data mapping errors that have occurred.
- If some or no indicator variables are provided, all rows are returned as above until the first data mapping error that does not have indicator variables is detected. The rows successfully fetched are returned and the SQLSTATE, SQLCODE, and SQLWARN flags are set, if necessary. (The SQLCODE may be 0 or a positive value).

It is possible, if a data mapping error occurs, for the positioning of the cursor to be successful. In this case, the cursor is positioned on the rowset that encountered the data mapping error.

Consider the following examples, which try to fetch 10 rows with a single FETCH statement.

- **Example 1:** Assume that indicators have been provided for values returned for column 1, but not for column 2. The 5th row has a data mapping error (+802) for column 1, and the 7th row has a data mapping error for column 2 (-802 is returned because an indicator was not provided for column 2). SQLERRD3 is set to 6 for the 6 returned rows, SQLSTATE and SQLCODE are set to the error from the 7th row fetched. The indicator variable for the 5th row column 1 indicates that a data mapping error was found. This information is also available from the GET DIAGNOSTICS statement, for example:

  ```
  GET DIAGNOSTICS :num_rows = ROW_COUNT, :num_cond = NUMBER;
  -- Results of the statement:
  -- num_rows = 6 and num_cond = 2 (2 conditions)
  GET DIAGNOSTICS CONDITION 1 :sqlstate = RETURNED_SQLSTATE,
  :sqlcode = DB2_RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;
  -- Results of the statement:
  -- sqlstate = 01519, sqlcode = +802, and row_num = 5
  GET DIAGNOSTICS CONDITION 2 :sqlstate = RETURNED_SQLSTATE,
  :sqlcode = DB2_RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;
  -- Results of the statement:
  -- sqlstate = 22003, sqlcode = -802, and row_num = 7
  ```

  The resulting cursor position is unknown.

- **Example 2:** Assume that null indicators are provided, that rows 3 and 5 are holes, and that data exists for the other requested rows. SQLERRD3 is set to 10 to reflect that 10 fetches were completed and that information has been returned for the 10 requested rows. Eight rows actually contain data. For two rows, indicator variables are set to indicate no data was returned for those rows. SQLSTATE is set to 02502, SQLCODE is set to +222, and all null indicators for rows 3 and 5 are set to -3 to indicate that a hole was detected. This information is also available from the GET DIAGNOSTICS statement, for example:

```
GET DIAGNOSTICS :num_rows = ROW_COUNT, :num_cond = NUMBER;
-- Results of the statement:
-- num_rows = 10 and num_cond = 2 (2 conditions)

GET DIAGNOSTICS CONDITION 1 :sqlstate = RETURNED_SQLSTATE,
:sqlcode = DB2_RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;
-- Results of the statement:
-- sqlstate = 02502, sqlcode = +222, and row_num = 3

GET DIAGNOSTICS CONDITION 2 :sqlstate = RETURNED_SQLSTATE,
:sqlcode = DB2_RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;
-- Results of the statement:
-- sqlstate = 02502, sqlcode = +222, and row_num = 5
```

  If a null indicator was not provided for any variable in a row that was a hole, an error occurs.

SQLCA usage summary:

For multiple-row-fetch, the fields of the SQLCA are set as follows:

Condition		Action: Resulting Values Stored in the SQLCA Fields		
Errors	**Data**	**SQLSTATE**	**SQLCODE**	**SQLERRD3**
No[1]	Return all requested rows	00000	0	Number of rows requested
No[1]	Return data for subset of requested rows, end of data	02000	+100	Number of rows rows
No[1]	Return all requested rows	sqlstate(2)	sqlcode(2)	Number of rows requested
Yes[1]	Return successfully fetched rows	sqlstate(3)	sqlcode(3)	Number of rows
Yes[1]	Return successfully fetched rows	sqlstate(4)	sqlcode(4)	Number of rows

Notes:

1. SQLWARN flags may be set in all cases, even if there are no other warnings or errors indicated. The warning flags are an accumulation of all warning flags set while processing the multiple-row-fetch.
2. sqlcode is the last positive SQLCODE, and sqlstate is the corresponding SQLSTATE value.
3. Database Server detected error. sqlcode is the first negative SQLCODE encountered, sqlstate is the corresponding SQLSTATE value.
4. Client detected error. sqlcode is the first negative SQLCODE encountered, sqlstate is one of the following SQLSTATEs: 22002, 22008, 22509, 22518, or 55021.

Providing indicator variables for error conditions:

If an error occurs as the result of an arithmetic expression in the SELECT list of an outer SELECT statement (division by zero or overflow) or a numeric conversion error occurs, the result is the null value. As in any

other case of a null value, an indicator variable must be provided and the main variable is unchanged. In this case, however, the indicator variable is set to -2. Processing of the statement continues as if the error had not occurred. (However, this error causes a positive SQLCODE.)

If you do not provide an indicator variable, a negative value is returned in the SQLCODE field of the SQLCA. Processing of the statement terminates when the error is encountered. No value is assigned to the host variable or to later variables, though any values that have already been assigned to variables remain assigned. Additionally, a -3 is returned in all indicators provided by the application when a hole was detected for the row on a rowset positioned FETCH, and values were not returned for the row. Processing of the statement terminates if a hole is detected and at least one indicator variable was not provided by the application.

Alternative syntax and synonyms:
USING DESCRIPTOR can be specified as a synonym for INTO DESCRIPTOR.

Example

Example 1: The FETCH statement fetches the results of the SELECT statement into the application program variables DNUM, DNAME, and MNUM. When no more rows remain to be fetched, the not found condition is returned.

```
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT DEPTNO, DEPTNAME, MGRNO FROM DSN8B10.DEPT
  WHERE ADMRDEPT = 'A00';
 EXEC SQL OPEN C1;
 DO WHILE (SQLCODE = 0);
  EXEC SQL FETCH C1 INTO :DNUM, :DNAME, :MNUM;
 END;
 EXEC SQL CLOSE C1;
```

Example 2: For an example of FETCH statements with a dynamic scrollable cursor, see Example 8.

Example 3: Fetch the last 5 rows of the result table C1 using cursor C1:

```
FETCH ROWSET STARTING AT ABSOLUTE -5
   FROM C1 FOR 5 ROWS INTO DESCRIPTOR :MYDESCR;
```

Example 4: Fetch 6 rows starting at row 10 for cursor CURS1, and fetch the data into three host-variable-arrays:

```
FETCH ROWSET STARTING AT ABSOLUTE 10
   FROM CURS1 FOR 6 ROWS
   INTO :hav1, :hva2, :hva3;
```

Alternatively, a descriptor could have been specified in an INTO DESCRIPTOR clause where the information in the SQLDA reflects the data types of the host-variable-arrays:

```
FETCH ROWSET STARTING AT ABSOLUTE 10
   FROM CURS1 FOR 6 ROWS
   INTO DESCRIPTOR :MYDESCR;
```

Example 5: Suppose that the following array type, array variable, and table have been defined.

```
CREATE TYPE INTARRAY AS INTEGER ARRAY[100];
CREATE TABLE T1 (COL1 CHAR(10), COL2 INT);
```

Use an array variable as an output target for a FETCH statement. The array variable is specified in the INTO clause of the FETCH statement.

```
CREATE PROCEDURE PROCESSINTARRAY (OUT INTOUTARRAY INTARRAY)
BEGIN
 DECLARE INTA INTARRAY;
 DECLARE INTB INTARRAY;
 DECLARE INTV INTEGER DEFAULT 1;
 DECLARE STMT CHAR(100);
 DECLARE C2 CURSOR FOR S1;
--
-- Initialize the array
--
 SET INTA = ARRAY[1,INTEGER(2),3+0,4,5,6] ;
--
-- Use dynamic SQL with an array parameter marker and a parameter marker
-- containing the index to retrieve the value from the array parameter.
-- The array is referenced in a predicate.
--
 SET STMT = 'SELECT COL1 FROM T1 WHERE COL2 = CAST(? AS INTARRAY)[?]';
 PREPARE S1 FROM STMT;
 OPEN C2 USING INTA, INTV;
 FETCH C2 INTO INTB ; -- INTB is an array variable that is used
                      -- as a target for the fetch statement.
 CLOSE C2;
SET INTOUTARRAY=INTB;
END
```

FREE LOCATOR

The FREE LOCATOR statement removes the association between a LOB locator variable and its value.

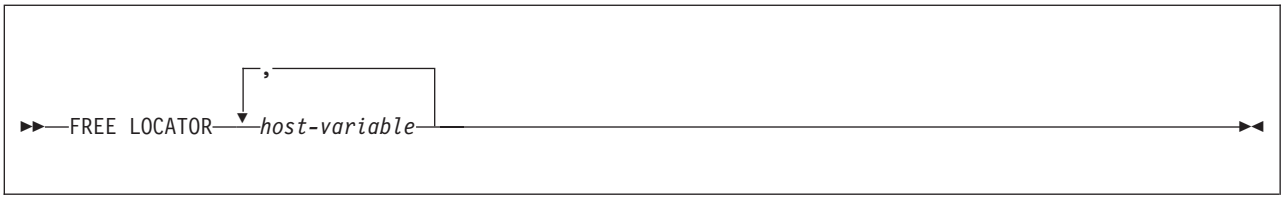
Invocation

This statement can only be embedded in an application program. It cannot be issued interactively. It is an executable statement that can be dynamically prepared. However, the EXECUTE statement with the USING clause must be used to execute the prepared statement. FREE LOCATOR cannot be used with the EXECUTE IMMEDIATE statement. It must not be specified in Java.

Authorization

None required.

Syntax



Description

host-variable, ...
Identifies one or more locator variables that must be declared in accordance with the rules for declaring locator variables. The locator variable type must be a binary large object locator, a character large object locator, or a double-byte character large object locator.

The *host-variable* must currently have a locator assigned to it. That is, a locator must have been assigned during this unit of work (by a FETCH, SELECT INTO, assignment statement, SET *host-variable* statement, or VALUES INTO statement) and must not subsequently have been freed (by a FREE LOCATOR statement); otherwise, an error is returned.

If more than one locator is specified and an error is returned on one of the locators, it is possible that some locators have been freed and others have not been freed.

Example

Assume that the employee table contains columns RESUME, HISTORY, and PICTURE and that locators have been established in a program to represent the column values. Free the CLOB locator variables LOCRES and LOCHIST, and the BLOB locator variable LOCPIC.

```
EXEC SQL FREE LOCATOR :LOCRES, :LOCHIST, :LOCPIC
```

GET DIAGNOSTICS

The GET DIAGNOSTICS statement provides diagnostic information about the last SQL statement (other than a GET DIAGNOSTICS statement) that was executed. This diagnostic information is gathered as the previous SQL statement is executed. Some of the information available through the GET DIAGNOSTICS statement is also available in the SQLCA.

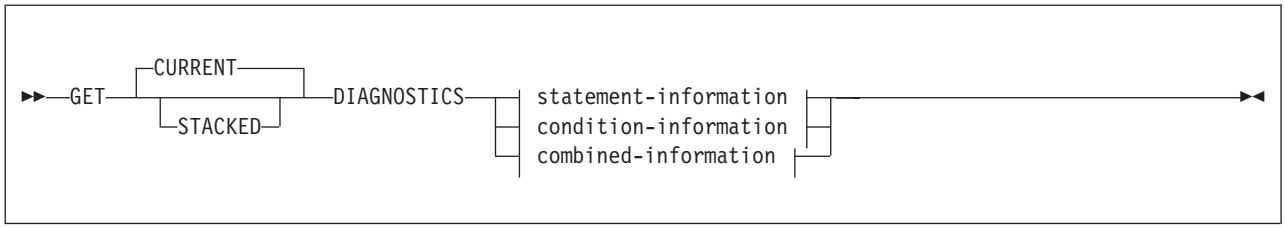
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

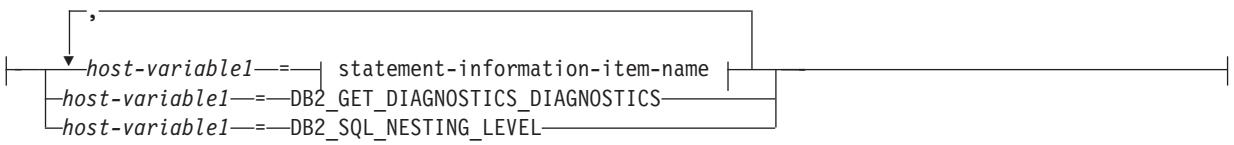
None required.

Syntax

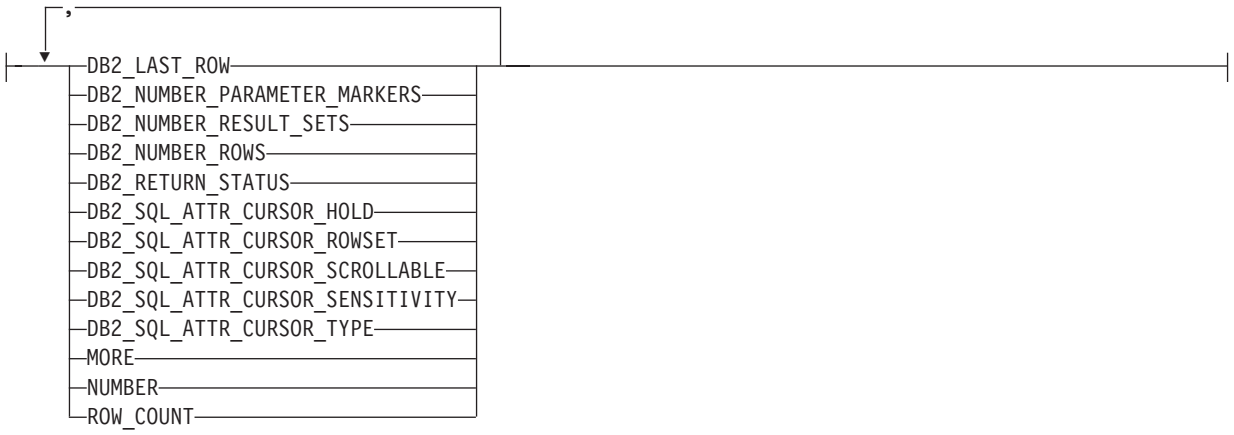


statement-information:

statement-information:

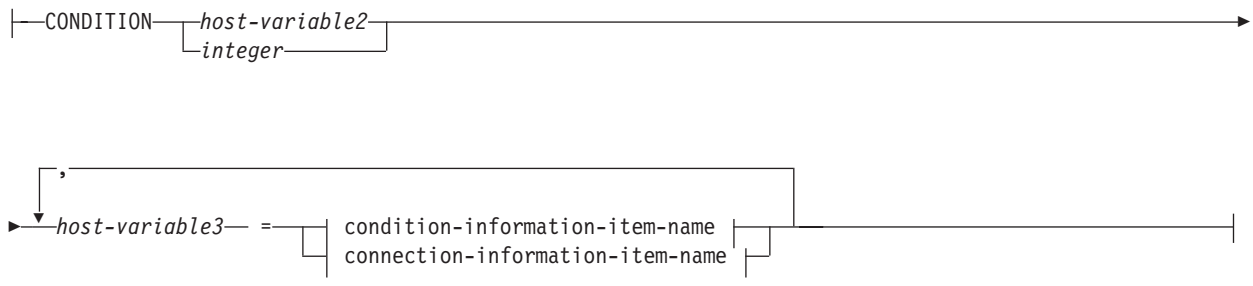


statement-information-item-name:

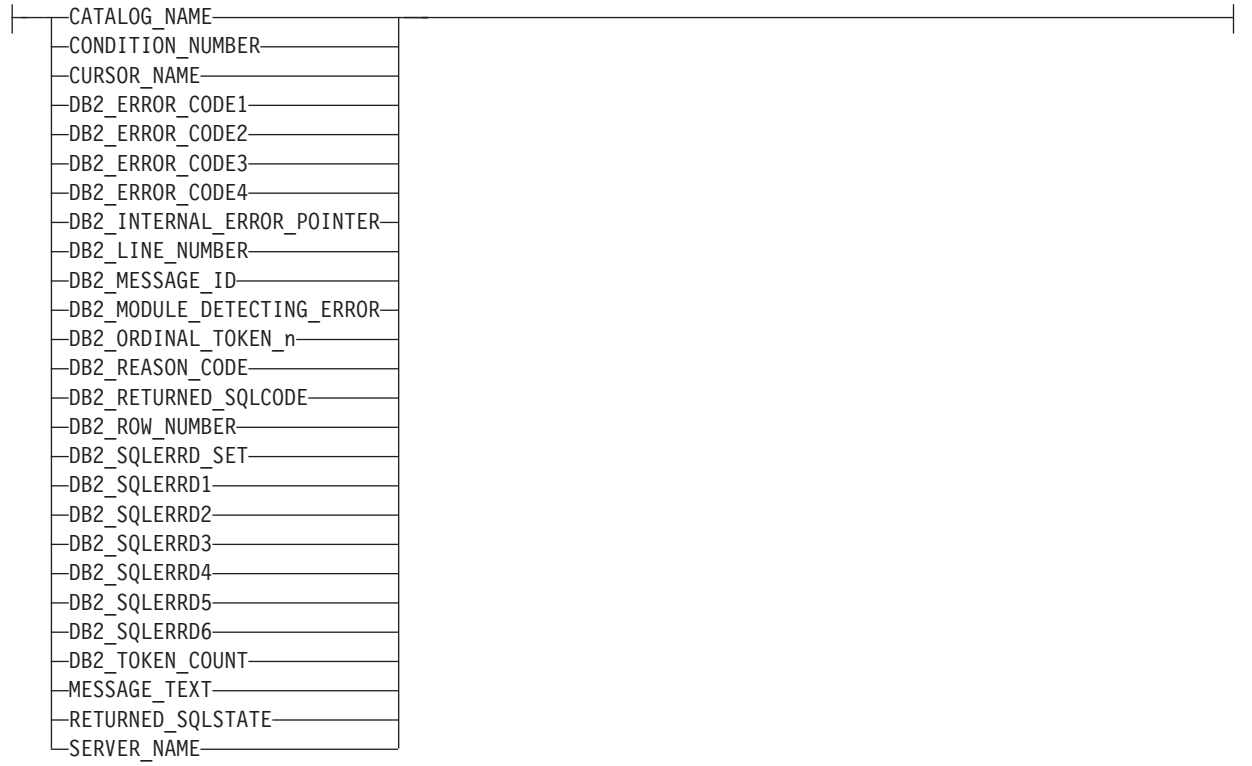


condition-information:

condition-information:



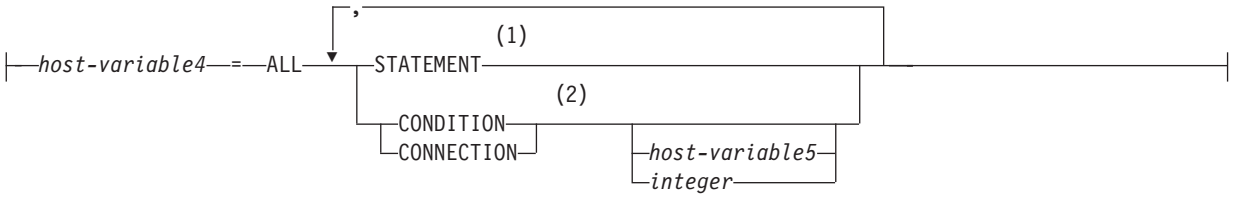
condition-information-item-name:



connection-information-item-name:

combined-information:

combined-information:



Notes:

1 STATEMENT can only be specified once.

2 CONDITION and CONNECTION can only be specified once if host-variable5 or integer is not also specified.

Description

Diagnostic information is provided in three main areas: statement information, condition information, and combined information. After the execution of an SQL statement, information about the execution of the statement is provided as statement information, and at least one instance of condition information is provided. The number of instances of the condition information is indicated by the NUMBER item that is available in the statement information. Combined information contains a text representation of all the information gathered about the execution of the SQL statement.

The diagnostic information that is provided is specific to the server. If you are connected to a server other than DB2 for z/OS, see that product's documentation for the diagnostic information that is returned.

CURRENT
Specifies that information is to be returned from the first diagnostics area. It corresponds to the previous SQL statement that was executed that was not a GET DIAGNOSTICS or compound statement. **CURRENT** is the default.

STACKED
Specifies that information is to be returned from the stacked diagnostics area. The stacked diagnostics area is only available within a handler in a native SQL procedure and non-inline SQL functions. The stacked diagnostics area corresponds to the previous SQL statement (that was not a GET DIAGNOSTICS or compound statement) that was executed before the handler was entered. If the GET DIAGNOSTICS statement is the first statement within a handler, the current diagnostics area and the stacked diagnostics area contain the same diagnostics information.

statement-information
Provides information about the last SQL statement executed.

host-variable1
Identifies a variable described in the program in accordance with the rules for declaring host variables. The data type of the host variable must be the data type as specified in Data types for GET DIAGNOSTICS items.

The host variable is assigned the value of the specified statement information item. If the value is truncated when assigning it to the host variable, a warning

is returned and the GET_DIAGNOSTICS_DIAGNOSTICS item of the diagnostics area is updated with the details of this condition. If a DIAGNOSTICS item is not set, then the host variable is set to a default value, based on its data type: 0 for an exact numeric field, an empty string for a VARCHAR field, and blanks for a CHAR field.

DB2_GET_DIAGNOSTICS_DIAGNOSTICS
Contains textual information about errors or warnings that might have occurred in the execution of the GET DIAGNOSTICS statement. The format of the information is similar to what would be returned by a GET DIAGNOSTICS :hv = ALL statement.

DB2_SQL_NESTING_LEVEL
Identifies the current level of nesting or recursion that is in effect when the GET DIAGNOSTICS statement was executed. Each level of nesting corresponds to a nested or recursive invocation of a packaged SQL function, packaged SQL procedure, or trigger. If the GET DIAGNOSTICS statement is executed outside of a level of nesting, the value of zero is returned.

statement-information-item-name:

DB2_LAST_ROW
For a multiple-row FETCH statement, contains a value of +100 if the last row currently in the table is in the set of rows that have been fetched. For cursors that are not sensitive to updates, there would be no need to do a subsequent FETCH, because the result would be an end-of-data indication. For cursors that are sensitive to updates, a subsequent FETCH may return more data if a row had been inserted before the FETCH was executed. For statements other than multiple-row FETCH statements, or for multiple-row FETCH statements that do not contain the last row, this variable contains the value 0.

An end of data warning might not occur and DB2_LAST_ROW might not contain +100 when the number of rows returned is equal to the number of rows requested and the last row of data returned is the last row of data.

DB2_NUMBER_PARAMETER_MARKERS
For a PREPARE statement, contains the number of parameter markers in the prepared statement. Otherwise, or if the server only returns an SQLCA, the value zero is returned.

DB2_NUMBER_RESULT_SETS
For a CALL statement, contains the actual number of result sets returned by the procedure. Otherwise, or if the server only returns an SQLCA, the value zero is returned.

DB2_NUMBER_ROWS
If the previous SQL statement was an OPEN or a FETCH that caused the size of the result table to be known, returns the number of rows in the result table. For SENSITIVE DYNAMIC cursors, this value can be thought of as an approximation because rows that are inserted and deleted will affect the next retrieval of this value. If the previous SQL statement was a PREPARE statement, returns the estimated number of rows in the result table for the prepared statement. Otherwise, or if the server only returns an SQLCA, the value zero is returned.

DB2_RETURN_STATUS
Identifies the status value returned from the stored procedure associated with the previously executed SQL statement, provided that the statement

was a CALL statement that invoked a procedure that returns a status. Otherwise, or if the server only returns an SQLCA, the value zero is returned.

DB2_SQL_ATTR_CURSOR_HOLD
For an ALLOCATE or OPEN statement, indicates whether a cursor can be held open across multiple units of work.

- N indicates that this cursor does not remain open across multiple units of work.
- Y indicates that this cursor remains open across multiple units of work.

Otherwise, a blank is returned.

DB2_SQL_ATTR_CURSOR_ROWSET
For an ALLOCATE or OPEN statement, indicates whether or not a cursor can be accesses using rowset positioning.

- N indicates that this cursor supports only row positioned operations.
- Y indicates that this cursor supports rowset positioned operations.

Otherwise, a blank is returned.

DB2_SQL_ATTR_CURSOR_SCROLLABLE
For an ALLOCATE or OPEN statement, indicates whether or not a cursor can be scrolled forward and backward.

- N indicates that this cursor is not scrollable.
- Y indicates that this cursor is scrollable.

Otherwise, a blank is returned.

DB2_SQL_ATTR_CURSOR_SENSITIVITY
For an ALLOCATE or OPEN statement, indicates whether or not a cursor does or does not show updates to cursor rows made by other connections.

- I indicates insensitive.
- S indicates sensitive.

Otherwise, a blank is returned.

DB2_SQL_ATTR_CURSOR_TYPE
For an ALLOCATE or OPEN statement, indicates the type of cursor, whether a cursor type is forward-only, static, or dynamic.

- F indicates a forward cursor.
- D indicates a dynamic cursor.
- S indicates a static cursor.

Otherwise, a blank is returned.

MORE
Indicates whether some of the warning and errors from the previous SQL statement were stored or discarded.

- N indicates that all the warnings and errors from the previous SQL statement are stored in the diagnostic area.
- Y indicates that some of the warnings and errors from the previous SQL statement were discarded because the amount of storage needed to record warnings and errors exceeded 65535 bytes.

NUMBER
Returns the number of errors and warnings detected by the execution of the previous SQL statement, other than a GET DIAGNOSTICS statement, that have been stored in the diagnostics area. If the previous SQL

statement returned an SQLSTATE of 00000 or no previous SQL statement has been executed, the number returned is one.

The GET DIAGNOSTICS statement itself may return information via the SQLSTATE parameter, but does not modify the previous contents of the diagnostics area, except for the DB2_GET_DIAGNOSTICS_DIAGNOSTICS item.

ROW_COUNT
Identifies the number of rows associated with the previous SQL statement that was executed.

If the previous SQL statement is a DELETE, INSERT, UPDATE, or MERGE statement, ROW_COUNT indicates the number of rows that are qualified to be deleted, inserted, or updated by that statement, excluding rows that are affected by triggers or referential integrity constraints. The count does not include rows that are inserted as a result of processing a FOR PORTION OF clause for in an SQL data change statement.

For the OPEN of a cursor for a SELECT with a data change statement, or a SELECT INTO statement, SQLERRD(3) contains the number of rows affected by the embedded data change statement. The value is 0 if the SQL statement fails, indicating that all changes made in executing the statement canceled.

A value of -1 indicates a mass delete from a table in a segmented table space and the DELETE statement did not include selection criteria, or a truncate operation. If the delete was against a view, then neither the DELETE statement nor the nor the definition of the view included selection criteria.

For a REFRESH TABLE statement, SQLERRD(3) contains the number of rows inserted into the materialized query table.

If the previous SQL statement is a multiple-row FETCH, ROW_COUNT identifies the number of rows fetched.

Otherwise, or if the server only returns an SQLCA, the value zero is returned.

condition-information
Assigns the values of the specified condition information to the associated host variables. The host variable specified must be of the data type that is compatible with the data type of the specified diagnostic-ID or an error occurs. If the value of the condition is truncated when assigning it to the host variable, an error occurs. If an indicator variable was provided, the length of the value is returned in the indicator variable.

If a DIAGNOSTICS item is not set, then the host variable is set to a default value, based on the data type of the item. The specific value will be 0 for a numeric field, an empty string for a VARCHAR field, and blanks for a CHAR field.

host-variable2 **or integer**
Identifies the diagnostic for which information is requested. Each diagnostic that occurs while executing an SQL statement is assigned an integer. The value 1 indicates the first diagnostic, 2 indicates the second diagnostic, and so on. If the value is 1, the diagnostic information that is retrieved corresponds to the condition that is indicated by the SQLSTATE value actually returned by the execution of the previous SQL statement (other than a GET DIAGNOSTICS statement). The host variable specified must be an integer data type or an error

occurs. An indicator variable is not allowed for this host variable. If a value is specified that is less than or equal to zero or greater than the number of available diagnostics, an error occurs.

host-variable3
Identifies a variable described in the program in accordance with the rules for declaring host variables. The data type of the host variable must be the data type as specified in Data types for GET DIAGNOSTICS items for the indicated condition-information item.

condition-information-item-name

CATALOG_NAME
If the returned SQLSTATE is any one of the following values, the constraint that caused the error is a referential, check, or unique constraint. The location (RDB) name of the server that generated the condition is returned.
- Class 09 (Triggered Action Exception),
- Class 23 (Integrity Constraint Violation)
- Class 27 (Triggered Data Change Violation)
- 40002 (Transaction Rollback - Integrity Constraint Violation)
- 40004 (Transaction Rollback - Triggered Action Exception)

If the returned SQLSTATE is class 42 (Syntax Error or Access Rule Violation), the server name of the table that caused the error is returned.

If the returned SQLSTATE is class 44 (WITH CHECK OPTION Violation), the server name of the view that caused the error is returned.

Otherwise, the empty string is returned.

The actual server name may be different than the server name specified, either implicitly or explicitly, on the CONNECT statement because of the use of aliases or synonyms.

CONDITION_NUMBER
Returns the number of the diagnostic returned.

CURSOR_NAME
If the returned SQLSTATE is class 24 (Invalid Cursor State), the name of the cursor is returned. Otherwise, the empty string is returned.

DB2_ERROR_CODE1
Returns an internal error code. Otherwise, or if the server only returns an SQLCA, the value 0 is returned.

DB2_ERROR_CODE2
Returns an internal error code. Otherwise, or if the server only returns an SQLCA, the value 0 is returned.

DB2_ERROR_CODE3
Returns an internal error code. Otherwise, or if the server only returns an SQLCA, the value 0 is returned.

DB2_ERROR_CODE4
Returns an internal error code. Otherwise, or if the server only returns an SQLCA, the value 0 is returned.

DB2_INTERNAL_ERROR_POINTER
For some errors, this is a negative value that is an internal error pointer. Otherwise, the value 0 is returned.

DB2_LINE_NUMBER
Returns the line number where an error is encountered in parsing a dynamic statement. Also returns the line number where an error is encountered in parsing, binding, or executing a CREATE or ALTER statement for a native SQL procedure. DB2_LINE_NUMBER also returns the line number when a CALL statement invokes a native SQL procedure and the procedure returns with an error. This information is not returned for an external SQL procedure.

This value will only be meaningful if the statement source contains new line control characters.

DB2_MESSAGE_ID
Corresponds to the message that is contained in the MESSAGE_TEXT diagnostic item (for example, DSNT102I or DSNU180I).

DB2_MODULE_DETECTING_ERROR
Returns an identifier indicating which module detected the error. For a SIGNAL statement that is issued from a routine, the value 'ROUTINE' is returned. Otherwise, the string 'DSN ' is returned.

DB2_ORDINAL_TOKEN_n
Returns the *n*th token. *n* must be a value from 1 to 100. For example, DB2_ORDINAL_TOKEN_1 would return the value of the first token, DB2_ORDINAL_TOKEN_2 the second token, and so on. A numeric value for a token is converted to characters before being returned. If there is no value for the token, or if the server only returns an SQLCA, an empty string is returned.

DB2_REASON_CODE
Contains the reason code for errors that have a reason code token in the message text. Otherwise, the value zero is returned.

DB2_RETURNED_SQLCODE
Returns the SQLCODE for the specified diagnostic.

DB2_ROW_NUMBER
Returns the number of the row where the condition was encountered, when such information is available and applicable. If SQLCODE +1– or +20237 is returned, DB2_ROW_NUMBER returns a value of 0.

DB2_SQLERRD_SET
A value of Y indicates that the DB2_SQLERRD1 through DB2_SQLERRD items might be set. These items are set only when communicating with a server that returns the SQLCA SQL communications area and not the new diagnostics area. Otherwise, a blank is returned.

DB2_SQLERRD1
Returns the value of sqlerrd(1) from the SQLCA that is returned by the server. Otherwise, the value zero is returned.

DB2_SQLERRD2
Returns the value of sqlerrd(2) from the SQLCA that is returned by the server. Otherwise, the value zero is returned.

DB2_SQLERRD3
Returns the value of sqlerrd(3) from the SQLCA that is returned by the server. Otherwise, the value zero is returned.

DB2_SQLERRD4
Returns the value of sqlerrd(4) from the SQLCA that is returned by the server. Otherwise, the value zero is returned.

DB2_SQLERRD5
Returns the value of sqlerrd(5) from the SQLCA that is returned by the server. Otherwise, the value zero is returned.

DB2_SQLERRD6
Returns the value of sqlerrd(6) from the SQLCA that is returned by the server. Otherwise, the value zero is returned.

DB2_TOKEN_COUNT
Returns the number of tokens available for the specified diagnostic ID.

MESSAGE_TEXT
Returns the message text that is associated with the SQLCODE. This is the short text, including substituted tokens. The message text does not contain the message number. When the SQLCODE is 0, the empty string is returned, even if the RETURNED_SQLSTATE value indicates a warning condition.

RETURNED_SQLSTATE
Returns the SQLSTATE for the specified diagnostic.

SERVER_NAME
If the previous SQL statement is a CONNECT, DISCONNECT, or SET CONNECTION statement, returns the name of the server specified in the previous statement is returned. Otherwise, the name of the server where the statement executes is returned.

connection-information-item-name
Provides information about the last SQL statement executed if it was a CONNECT statement.

DB2_AUTHENTICATION_TYPE
Contains an authentication type value of:

- 'S' for a server authentication
- 'C' for client authentication
- 'T' for trusted server authentication
- Otherwise, or if the server only returns an SQLCA, a blank is returned

DB2_AUTHORIZATION_ID
Authorization ID used by connected server. Because of user ID translation and authorization exits, the local user ID may not be the authorized ID used by the server.

DB2_CONNECTION_STATE
Contains the connection state:

- -1 if the connection is unconnected
- 1 if the connection is connected

Otherwise, or if the server only returns an SQLCA, the value zero is returned.

DB2_CONNECTION_STATUS
Contains a value of:

- 1 if committable updates can be performed on the connection for this unit of work
- 2 if no committable updates can be performed on the connection for this unit of work

Otherwise, or if the server only returns an SQLCA, the value zero is returned.

DB2_SERVER_CLASS_NAME
For a CONNECT or SET CONNECTION statement, contains one of the following values:

- QAS for DB2 for i
- QDB2 forDB2 for z/OS
- QDB2/2 for DB2 for OS/2
- QDB2/6000 for DB2 for AIX®
- QDB2/6000 PE for DB2 for AIX Parallel Edition
- QDB2/AIX64 for DB2 for AIX 64-bit
- QDB2/HPUX for DB2 for HP-UX
- QDB2/HP64 for DB2 for HP-UX 64-bit
- QDB2/LINUX for DB2 for Linux, UNIX, and Windows
- QDB2/LINUX390 for DB2 for Linux, UNIX, and Windows
- QDB2/LINUXIA64 for DB2 for Linux, UNIX, and Windows
- QDB2/LINUXPPC forDB2 for Linux, UNIX, and Windows
- QDB2/LINUXPPC64 for DB2 for Linux, UNIX, and Windows
- QDB2/LINUXZ64 for DB2 for Linux, UNIX, and Windows
- QDB2/NT for DB2 for Linux, UNIX, and Windows
- QDB2/NT64 for DB2 for Linux, UNIX, and Windows
- QDB2/PTX for DB2 for NUMA-Q®
- QDB2/SCO for DB2 for SCO UnixWare
- QDB2/SGI for DB2 for Silicon Graphics
- QDB2/SNI for DB2 for Siemens Nixdorf
- QDB2/SUN for DB2 for SUN Solaris
- QDB2/SUN64 for DB2 for SUN Solaris 64-bit
- QDB2/Windows 95 for DB2 for Linux, UNIX, and Windows
- QSQLDS/VM for DB2 Server for VSE & VM
- QSQLDS/VSE for DB2 Server for VSE & VM

Otherwise, the empty string is returned.

DB2_ENCRYPTION_TYPE
The level of encryption for the connection:

- A indicates only the authentication tokens (authid and password) are encrypted.
- D indicates all data is encrypted for the connection.
- Otherwise, a blank is returned.

DB2_PRODUCT_ID
Returns a product signature. If the application server is an IBM relational database product, the form is *pppvvrrm*, where:

- *ppp* identifies the product as follows:
  - ARI for DB2 Server for VSE & VM
  - DSN for DB2 for z/OS
  - QSQ for DB2 for i
  - SQL for all other DB2 products
- *vv* is a two-digit version identifier such as '11'
- *rr* is a two-digit release identifier such as '01'
- *m* is a one-digit modification level.

- Values 0, 1, 2, 3, and 4 are reserved for modification levels in conversion and enabling-new-function mode from Version 10 (CM10, CM10*, ENFM10, and ENFM10*)
- Values 5, 6, 7, 8, and 9 are for modification levels in new-function mode.

For example, if the application server is Version 9 of DB2 for z/OS in new-function mode with the latest maintenance, the value would be 'DSN09015'.

combined-information

Provides a text representation of all the information gathered about the execution of the SQL statement.

ALL

Indicates that all diagnostic items that are set for the last SQL statement executed are to be combined into one string. The format of the string is a semicolon separated list of all of the available diagnostic information in the form: <item-name>[(<condition-number>)]=<value-converted-to-character>;... as shown in the following example:

```
NUMBER=1;RETURNED_SQLSTATE=02000;DB2_RETURNED_SQLCODE=+100;
```

host-variable4

Identifies a variable described in the program in accordance with the rules for declaring host variables. The data type of the host variable must be VARCHAR. If the length of *host-variable4* is not sufficient to hold the full returned diagnostic string, the string is truncated, a warning is returned, and the GET_DIAGNOSTICS_DIAGNOSITICS item of the diagnostics area is updated with the details of this condition.

STATEMENT

Indicates that all statement-information-item-name diagnostic items that are set for the last SQL statement executed should be combined into one string. The format is the same as described for the ALL option.

CONDITION

Indicates that all condition-information-item-name diagnostic items that are set for the last SQL statement executed should be combined into one string. If *host-variable5* or integer is supplied after CONDITION, the format is the same as described above for the ALL option. If *host-variable5* or integer is not supplied, the format includes a condition number entry at the beginning of the information for that condition in the form:

CONDITION_NUMBER=X;<item-name>=<value-converted-to-character>;... where X is the number of the condition, as shown in the following example:

```
CONDITION_NUMBER=1;RETURNED_SQLSTATE=02000;RETURNED_SQLCODE=100;
   CONDITION_NUMBER=2;RETURNED_SQLSTATE=01004;
```

CONNECTION

Indicates that all connection-information-item-name diagnostic items that are set for the last SQL statement executed should be combined into one string. If *host-variable5* or integer is supplied after CONNECTION, the format is the same as described for the ALL option. If *host-variable5* or integer is not supplied, then the format includes a condition number entry at the beginning of the information for that condition in the form:

CONNECTION_NUMBER=X;<item-name>=<value-converted-to-character>;... where X is the number of the condition, as shown in the following example:

```
CONNECTION_NUMBER=1;CONNECTION_NAME=SVL1;DB2_PRODUCT_ID=DSN08010;
```

host-variable5 **or** *integer*
Identifies the diagnostic for which ALL CONDITION or ALL CONNECTION information is requested. The host variable specified must be an integer data type or an error occurs. An indicator variable is not allowed for this host variable or an error occurs. If a value is specified that is less than or equal to zero or greater than the number of available diagnostics, an error occurs.

Notes

Effect of the statement in a native SQL procedure:
The GET DIAGNOSTICS statement does not change the contents of the diagnostics area except for DB2_GET_DIAGNOSTICS_DIAGNOSTICS.

If you want information about an error, the GET DIAGNOSTICS statement must be the first executable statement specified in the handler that will handle the error condition.

If you want information about a warning and a handler will get control for the warning condition, the GET DIAGNOSTICS statement must be the first executable statement specified in that handler.

If you want information about a warning and a handler will not get control for the warning condition, the GET DIAGNOSTICS statement must be the next statement executed after that previous statement.

Considerations for the SQLSTATE and SQLCODE SQL variables:
The GET DIAGNOSTICS statement does not change the value of the SQLSTATE and SQLCODE SQL variables.

Data types for items:
When a diagnostic item is assigned to a host variable, SQL variable, or SQL parameter, the data type of the target must be compatible with the data type of the requested diagnostic item.

Data types for GET DIAGNOSTICS items

Table 145. Data types for GET DIAGNOSTICS items

Type of information	Item	Data type
Statement Information	DB2_GET_DIAGNOSTICS_DIAGNOSTICS	VARCHAR(32672)
	DB2_LAST_ROW	INTEGER
	DB2_NUMBER_PARAMETER_MARKERS	INTEGER
	DB2_NUMBER_RESULT_SETS	INTEGER
	DB2_NUMBER_ROWS	DECIMAL(31,0)
	DB2_RETURN_STATUS	INTEGER
	DB2_SQL_ATTR_CURSOR_HOLD	CHAR(1)
	DB2_SQL_ATTR_CURSOR_ROWSET	CHAR(1)
	DB2_SQL_ATTR_CURSOR_SCROLLABLE	CHAR(1)
	DB2_SQL_ATTR_CURSOR_SENSITIVITY	CHAR(1)
	DB2_SQL_ATTR_CURSOR_TYPE	CHAR(1)
	MORE	CHAR(1)
	NUMBER	INTEGER
	ROW_COUNT	DECIMAL(31,0)

Table 145. Data types for GET DIAGNOSTICS items (continued)

Type of information	Item	Data type
Statement Information	DB2_SQL_NESTING_LEVEL	INTEGER
Condition Information	CATALOG_NAME	VARCHAR(128)
	CONDITION_NUMBER	INTEGER
	CURSOR_NAME	VARCHAR(128)
	DB2_ERROR_CODE1	INTEGER
	DB2_ERROR_CODE2	INTEGER
	DB2_ERROR_CODE3	INTEGER
	DB2_ERROR_CODE4	INTEGER
	DB2_INTERNAL_ERROR_POINTER	INTEGER
	DB2_LINE_NUMBER	INTEGER
	DB2_MESSAGE_ID	CHAR(10)
	DB2_MODULE_DETECTING_ERROR	CHAR(8)
	DB2_ORDINAL_TOKEN_n	VARCHAR(515)
	DB2_REASON_CODE	INTEGER
	DB2_RETURNED_SQLCODE	INTEGER
	DB2_ROW_NUMBER	DECIMAL(31,0)
	DB2_TOKEN_COUNT	INTEGER
	MESSAGE_TEXT	VARCHAR(32672)
	RETURNED_SQLSTATE	CHAR(5)
	SERVER_NAME	VARCHAR(128)
Connection Information	DB2_AUTHENTICATION_TYPE	CHAR(1)
	DB2_AUTHORIZATION_ID	VARCHAR(128)
	DB2_CONNECTION_STATE	INTEGER
	DB2_CONNECTION_STATUS	INTEGER
	DB2_ENCRYPTION_TYPE	CHAR(1)
	DB2_PRODUCT_ID	VARCHAR(8)
	DB2_SERVER_CLASS_NAME	CHAR(128)
Combined Information	ALL	VARCHAR(32672)

DRDA considerations

The GET DIAGNOSTICS statement is supported from a current DB2 for z/OS client, regardless of the level of the server (a DB2 for z/OS Version 7 or a DB2 for Windows Version 7, for example). When the application is connected to servers that do not support the Open Group Version 3 DRDA standard, the diagnostic information that is returned by the servers is available in the condition information.

Alternative syntax and synonyms:

To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords:

- RETURN_STATUS as a synonym for DB2_RETURN_STATUS
- EXCEPTION as a synonym for CONDITION

Examples

Example 1: In an application, use the GET DIAGNOSTICS statement to determine how many rows were updated.

```
long rcount;
EXEC SQL UPDATE T1 SET C1 = C1 + 1;
EXEC SQL GET DIAGNOSTICS :rcount = ROW_COUNT;
```

After execution of this code segment, *rcount* will contain the number of rows that were updated.

Example 2: In an application, use the GET DIAGNOSTICS statement to handle multiple SQL Errors.

```
long numerrors, counter;
char retsqlstate[5];
long hva[5];
EXEC SQL INSERT INTO T1 FOR 5 ROWS VALUES (:hva) NOT ATOMIC
  CONTINUE ON SQLEXCEPTION;
EXEC SQL GET DIAGNOSTICS :numerrors = NUMBER;
for ( i=1;i < numerrors;i++)
 {
 EXEC SQL GET DIAGNOSTICS CONDITION :i :retsqlstate = RETURNED_SQLSTATE;
...
```

Execution of this code segment sets and prints *retsqlstate* with the SQLSTATE for each error that was encountered in the previous SQL statement.

Example 3: Retrieve information about a connection.

```
EXEC SQL GET DIAGNOSTICS CONDITION :HV_PRODUCT_ID = DB2_PRODUCT_ID;
```

Example 4: Use the GET DIAGNOSTICS statement to retrieve information that is similar to what is returned in the SQLCA:

```
EXEC SQL GET DIAGNOSTICS CONDITION 1
     :dasqlcode   = DB2_RETURNED_SQLCODE,
     :datokencnt  = DB2_TOKEN_COUNT,
     :datoken1    = DB2_ORDINAL_TOKEN_1,
     :datoken2    = DB2_ORDINAL_TOKEN_2,
     :datoken3    = DB2_ORDINAL_TOKEN_3,
     :datoken4    = DB2_ORDINAL_TOKEN_4,
     :datoken5    = DB2_ORDINAL_TOKEN_5,
     :dasqlerrd1b = DB2_MESSAGE_ID,
     :damsgtext   = MESSAGE_TEXT,
     :dasqlerrp   = DB2_MODULE_DETECTING_ERROR,
     :dasqlstate  = RETURNED_SQLSTATE;
```

Example 5: Specify the **STACKED** keyword on a GET DIAGNOSTICS statement that is used within a handler to access information in the diagnostics area that caused the handler to be activated:

```
  CREATE PROCEDURE divide2 ( IN numerator INTEGER,
                             IN denominator INTEGER,
                            OUT divide_result INTEGER,
                             OUT divide_error VARCHAR(70))
              LANGUAGE SQL
    BEGIN
        DECLARE msg_text        CHAR(70) DEFAULT '';
        DECLARE divide_error    CHAR(70) DEFAULT '';

        DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
           BEGIN
              INSERT .....;   -- insert row into a log table
```

```
            -- get diagnostic information for the INSERT statement
          GET CURRENT DIAGNOSTICS CONDITION 1 msg_text = MESSAGE_TEXT;

            -- get information about condition that activated the handler
          GET STACKED DIAGNOSTICS CONDITION 1 divide_error = MESSAGE_TEXT;
        END;

      SET divide_result = numerator/denominator;
  END;
```

The first GET DIAGNOSTICS statement obtains diagnostic information about the INSERT statement.

The second GET DIAGNOSTICS statement specifies the STACKED keyword. The use of the STACKED keyword allows access the stacked diagnostics area which contains the diagnostic information for the condition that caused the handler to be activated. The information about the original condition is still accessible within the handler even after another statement has been issued, such as the INSERT statement in the example.

Example 6: The following application logs information whenever a routine is invoked directly by an application rather than indirectly by another routine. The application uses the GET DIAGNOSTICS statement that specifies DB2_SQL_NESTING_LEVEL to obtain the current nesting level, and invokes the LOG_INVOCATION procedure if the nesting level is 1:

```
CREATE OR REPLACE PROCEDURE TEST
 MODIFIES SQL DATA
 LANGUAGE SQL
 BEGIN
    DECLARE NESTING_LEVEL  INT     DEFAULT 0;

    GET DIAGNOSTICS NESTING_LEVEL = DB2_SQL_NESTING_LEVEL;

    --
    -- If routine is invoked at nesting level 1,
    -- invoke a routine to log the invocation.
    --
    IF (NESTING_LEVEL = 1) THEN
       CALL LOG_INVOCATION();
    END IF;

    --
    -- Remainder of procedure logic
    --
    ...
END
```

GRANT

The DB2 GRANT statement grants privileges to authorization IDs. There is a separate form of the statement for each of these classes of privilege:

- Collection
- Database
- Distinct type
- Function or stored procedure
- Package
- Plan
- Schema
- Sequence
- System
- Table or view
- Distinct type, array type, or JAR file
- Variable
- Use

The applicable objects are always at the current server. The grants are recorded in the current server's catalog.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

If the authorization mechanism was not activated when the DB2 subsystem was installed, an error condition occurs.

Authorization

To grant a privilege P, the privilege set must include one of the following:

- The privilege P WITH GRANT OPTION
- Ownership of the object on which P is a privilege
- SECADM authority

  Note: If installation parameter SEPARATE SECURITY is NO, SYSADM authority has implicit SECADM authority.
- ACCESSCTRL authority

The presence of ACCESSCTRL authority in the privilege set allows the granting of all authorities except:

- System DBADM
- CREATE_SECURE_OBJECT privilege
- DATAACCESS
- ACCESSCTRL

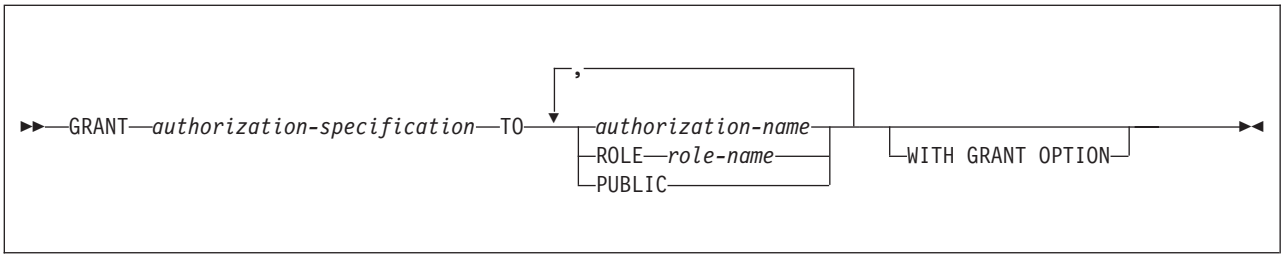
  Note: If installation parameter SEPARATE SECURITY is NO, SYSCTRL authority has implicit ACCESSCTRL authority that allows the granting of all privileges except:

- DBADM on databases
- DELETE, INSERT, SELECT, and UPDATE on user tables or views
- EXECUTE on plans, packages, functions, or stored procedures
- PACKADM on collections
- SYSADM authority
- USAGE on distinct types, JARs, and sequences
- READ, WRITE on global variables

Except for views, the GRANT option for privileges on a table is also inherent in DBADM authority for its database, provided DBADM authority was acquired with the GRANT option. See "CREATE VIEW" on page 1527 for a description of the rules that apply to views.

If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. The owner can be a role. If the statement is dynamically prepared, the privilege set is the privileges that are held by the SQL authorization ID of the process. However, if the process is running in a trusted context that is defined with the ROLE AS OBJECT OWNER CLAUSE, the privilege set is the privileges that are held by the role in effect.

Syntax



Description

authorization-specification
Specifies one or more privileges for the class of privilege. The same privilege must not be specified more than once.

TO Specifies to what authorization IDs the privileges are granted.

authorization-name,...
Lists one or more authorization IDs.

ROLE *role-name*
Lists one or more role names. Each name must identify a role that exists at the current server.

The value of the CURRENT RULES special register determines whether you can use the ID or role of the GRANT statement itself (to grant privileges to yourself). When CURRENT RULES is:
DB2 You cannot use the ID or role of the GRANT statement.
STD
You can use the ID or role of the GRANT statement.

PUBLIC
Grants the privileges to all users at the current server, including database requesters using DRDA access.

CREATE_SECURE_OBJECT must not be granted to PUBLIC.

ACCESSCTRL, DATAACCESS and system DBADM authorities cannot be granted to PUBLIC.

WITH GRANT OPTION

Allows the named users to grant the privileges to others. Granting an administrative authority with this option allows the user to specifically grant any privilege belonging to that authority. If you omit WITH GRANT OPTION, the named users cannot grant the privileges to others unless they have that authority from some other source.

GRANT authority cannot be passed to PUBLIC. When WITH GRANT OPTION is used with PUBLIC, a warning is issued, and the named privileges are granted, but without GRANT authority.

If you grant the CREATE_SECURE_OBJECT system privilege, the WITH GRANT OPTION clause is ignored because the CREATE_SECURE_OBJECT system privilege cannot be granted to others.

GRANT ACCESSCTRL, DATAACCESS and system DBADM authorities cannot be passed to others. If WITH GRANT OPTION is used when granting these authorities, a warning is issued and the named authorities are granted, but without GRANT authority.

Notes

For more on DB2 privileges, read *DB2 Administration Guide*.

A *grant* is the granting of a specific privilege by a specific grantor to a specific grantee. The grantor for a given GRANT statement is the authorization ID for the privilege set; that is, the SQL authorization ID of the process or a role, or the authorization ID of the owner of the plan or package. Grant statements that are made in a trusted context that is defined with the ROLE AS OBJECT OWNER clause result in the grantor being the role that is in effect. If the statement is prepared dynamically, the grantor is the role that is associated with the ID that is running the statement. If the statement is embedded in an application program that was bound in a trusted context that was defined with the ROLE AS OBJECT OWNER clause the owner of the plan or package is a role which is the grantor. If the ROLE AS OBJECT OWNER clause is not specified for the trusted context, the grantor is the authorization ID of the process.

The grantee, as recorded in the catalog, is an authorization ID or PUBLIC.

Duplicate grants from the same grantor are not recorded in the catalog. Otherwise, the result of executing a GRANT statement is recorded as one or more grants in the current server's catalog.

If more than one privilege or *authorization-name* is specified after the TO keyword and one of the grants is in error, execution of the statement is stopped and no grants are made. The status of the privilege or privileges granted is recorded in the catalog for each *authorization-name*.

Different grantors can grant the same privilege to a single grantee. The grantee retains that privilege as long as one or more of those grants are recorded in the catalog. Privileges that imply other privileges are also termed *authorities*. Grants are removed from the catalog by executing SQL REVOKE statements.

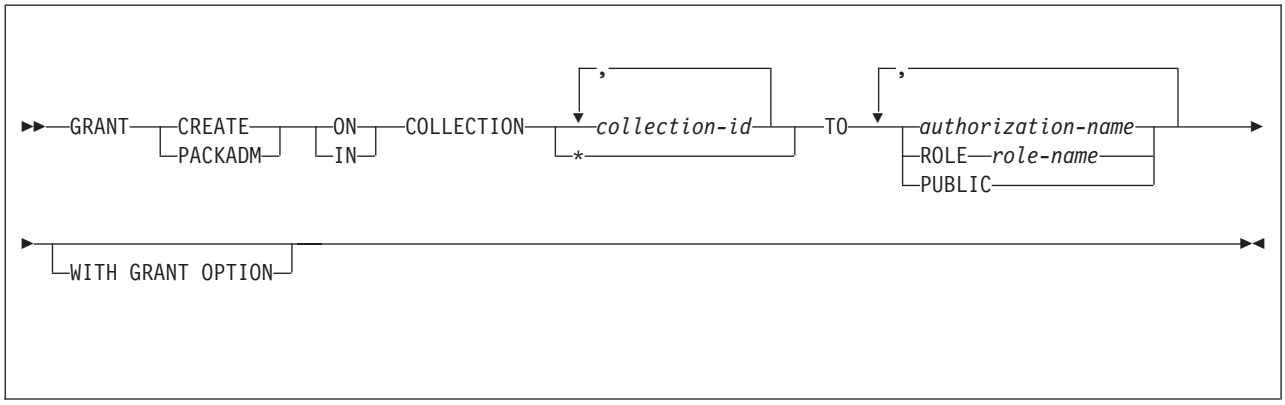
Whenever a grant is made for a database, distinct type, package, plan, schema, stored procedure, table, trigger, user-defined function, view, or USE privilege for an object that does not exist, an SQL return code is issued and the grant is not made.

The grantee, as recorded in the catalog for PUBLIC AT ALL LOCATIONS is PUBLIC*.

GRANT (collection privileges)

This form of the GRANT statement grants privileges on collections.

Syntax



Description

CREATE IN
Grants the privilege to use the BIND subcommand to create packages in the designated collections.

The word ON can be used instead of IN.

PACKADM ON
Grants package administrator authority for the designated collections.

The word IN can be used instead of ON.

COLLECTION *collection-id,...*
Identifies the collections on which the specified privilege is granted. The collections do not have to exist.

COLLECTION *
Indicates that the specified privilege is granted on all collections including those that do not currently exist.

TO Refer to "GRANT" on page 1695 for a description of the TO clause.

WITH GRANT OPTION
Refer to "GRANT" on page 1695 for a description of the WITH GRANT OPTION clause.

Examples

Example 1: Grant the privilege to create new packages in collections QAACLONE and DSN8CC61 to CLARK.

```
GRANT CREATE IN COLLECTION QAACLONE, DSN8CC61 TO CLARK;
```

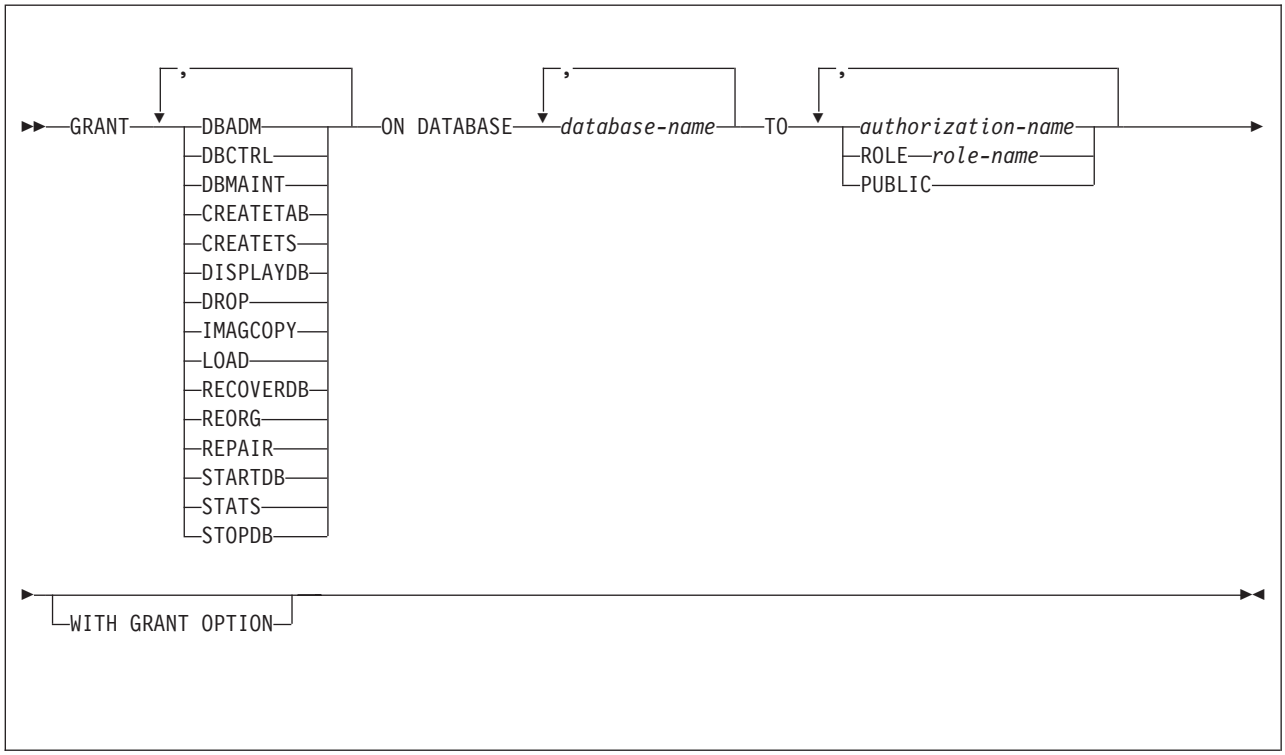
Example 2: Grant the privileges to create new packages in collection DSN8CC91 to role ROLE1:

```
GRANT CREATE IN COLLECTION DSN8CC91 TO ROLE ROLE1;
```

GRANT (database privileges)

This form of the GRANT statement grants privileges on databases.

Syntax



Description

Each keyword listed grants the privilege described, but only as it applies to or within the databases named in the statement.

DBADM
Grants the database administrator authority.

DBCTRL
Grants the database control authority.

DBMAINT
Grants the database maintenance authority.

CREATETAB
Grants the privilege to create new tables. To create tables in an implicitly created database, CREATETAB privileges are needed on the DSNDB04 database. For a work file database, PUBLIC implicitly has the CREATETAB privilege (without GRANT authority) to define declared temporary tables; this privilege is not recorded in the DB2 catalog, and it cannot be revoked.

CREATETS
Grants the privilege to create new table spaces.

DISPLAYDB
Grants the privilege to issue the DISPLAY DATABASE command.

DROP
Grants the privilege to issue the DROP or ALTER DATABASE statements for the designated databases.

IMAGCOPY
Grants the privilege to run the COPY, MERGECOPY, and QUIESCE utilities against table spaces of the specified databases, and to run the MODIFY RECOVERY utility.

LOAD
Grants the privilege to use the LOAD utility to load tables.

RECOVERDB
Grants the privilege to use the RECOVER and REPORT utilities to recover table spaces and indexes.

REORG
Grants the privilege to use the REORG utility to reorganize table spaces and indexes.

REPAIR
Grants the privilege to use the REPAIR and DIAGNOSE utilities.

STARTDB
Grants the privilege to issue the START DATABASE command.

STATS
Grants the privilege to use the RUNSTATS utility to update statistics, the CHECK utility to test whether indexes are consistent with the data they index, and the MODIFY STATISTICS utility to delete unwanted statistics history records from the corresponding catalog tables.

STOPDB
Grants the privilege to issue the STOP DATABASE command.

ON DATABASE *database-name,...*
Identifies databases on which privileges are to be granted. For each named database, the grantor must have all the specified privileges with the GRANT option. Each name must identify a database that exists at the current server. DSNDB01 must not be identified; however, a grant of a privilege on DSNDB06 implies the granting of the same privilege on DSNDB01 for utility operations only.

Database privileges granted on DSNDB04 are applicable to all implicitly created databases. This means that a user with the STOPDB privilege on DSNDB04 can also stop database objects in any implicitly created database. Similarly, having DBADM on DSNDB04 allows access to all tables in all implicitly created databases. However, having a database privilege on DSNDB04 does not allow granting of this privilege on an implicitly created database to others.

TO Refer to "GRANT" on page 1695 for a description of the TO clause.

WITH GRANT OPTION
Refer to "GRANT" on page 1695 for a description of the WITH GRANT OPTION clause.

Examples

Example 1: Grant drop privileges on database DSN8D11A to user PEREZ.

```
GRANT DROP
  ON DATABASE DSN8D11A
  TO PEREZ;
```

Example 2: Grant repair privileges on database DSN8D11A to all local users.

```
GRANT REPAIR
  ON DATABASE DSN8D11A
  TO PUBLIC;
```

Example 3: Grant authority to create new tables and load tables in database DSN8D11A to users WALKER, PIANKA, and FUJIMOTO, and give them grant privileges.

```
GRANT CREATETAB,LOAD
  ON DATABASE DSN8D11A
  TO WALKER,PIANKA,FUJIMOTO
  WITH GRANT OPTION;
```

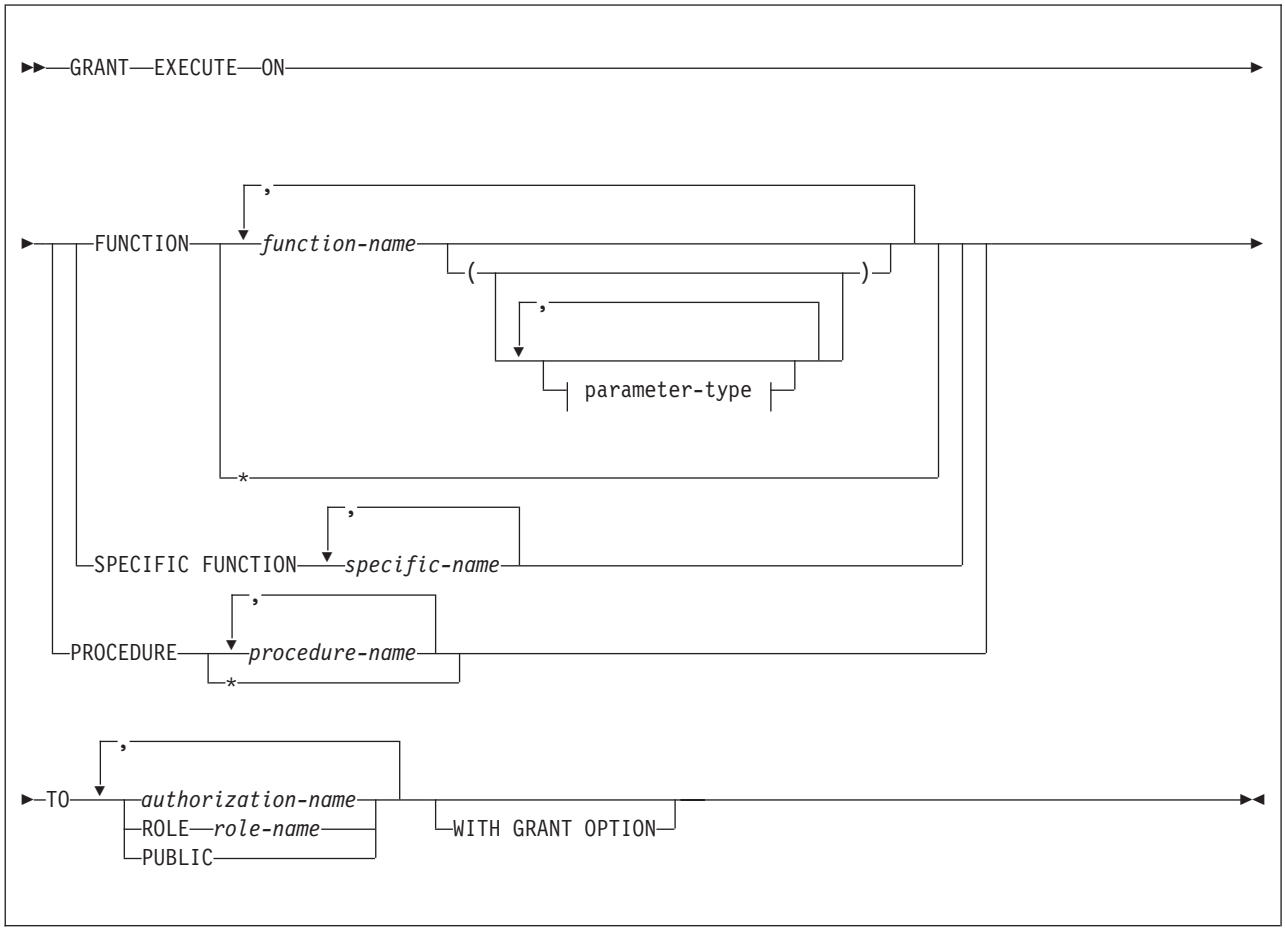
Example 4: Grant load privileges to database DSN9D91A to role ROLE1:

```
GRANT LOAD
  ON DATABASE DSN9D91A
  TO ROLE ROLE1;
```

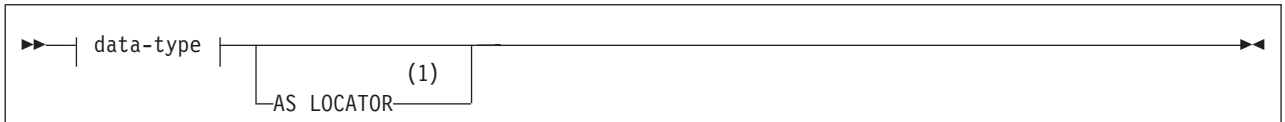
GRANT (function or procedure privileges)

This form of the GRANT statement grants privileges on user-defined functions, cast functions that are generated for distinct types, array types, and stored procedures.

Syntax



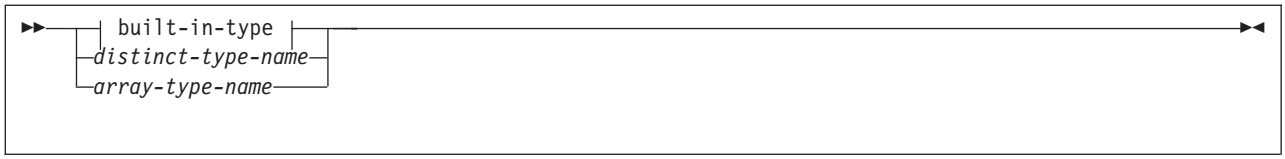
parameter-type:



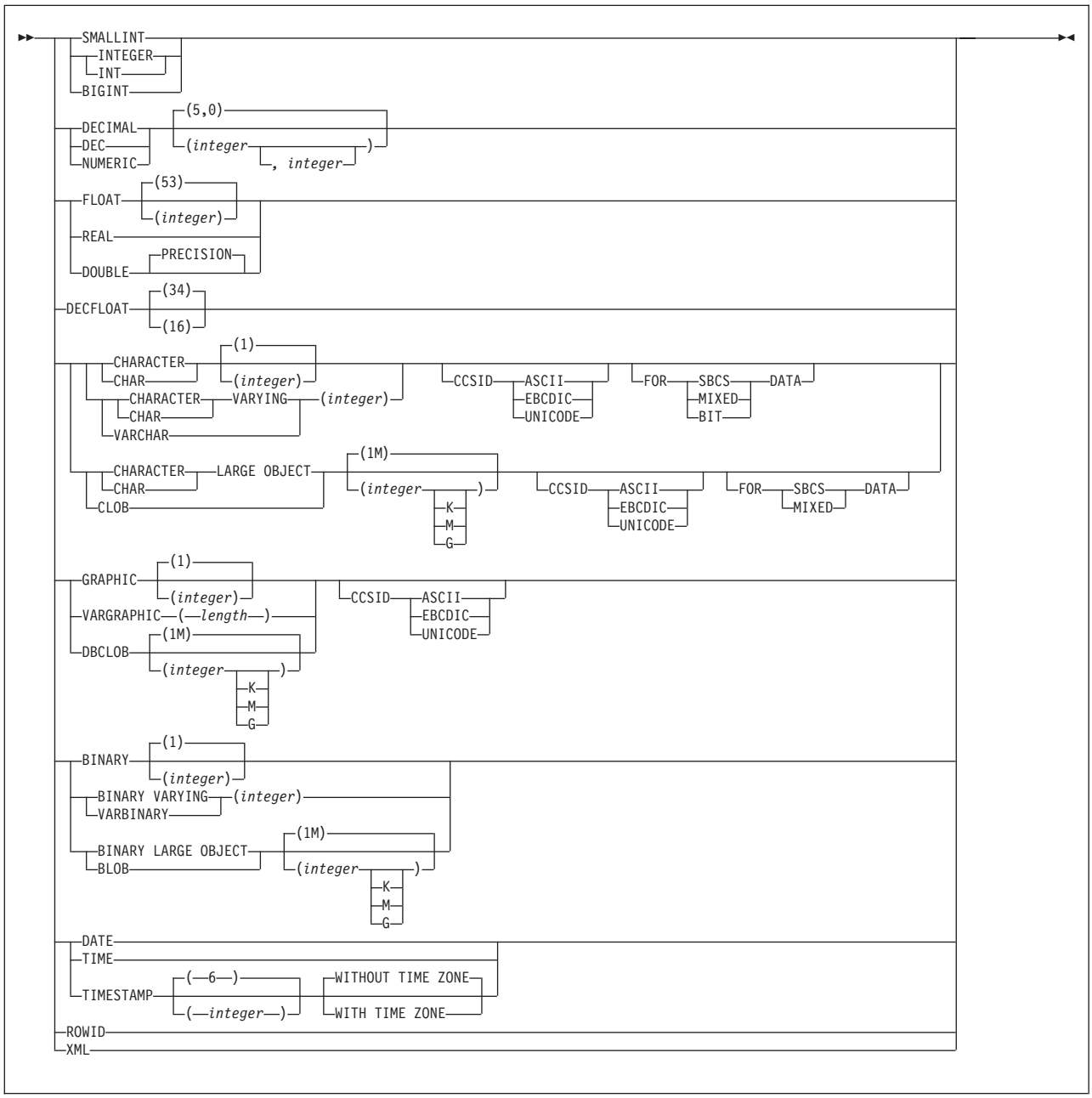
Notes:

1 AS LOCATOR can be specified only for a LOB data type or a distinct type based on a LOB data type.

data-type:



built-in-type:



Description

EXECUTE

Grants the privilege to run the identified user-defined function, cast function that was generated for a distinct type, or stored procedure.

FUNCTION or SPECIFIC FUNCTION

Identifies the function on which the privilege is granted. The function must exist at the current server, and it must be a function that was defined with the CREATE FUNCTION statement or a cast function that was generated by a CREATE TYPE statement. The function can be identified by name, function signature, or specific name.

If the function was defined with a table parameter (the LIKE TABLE was specified in the CREATE FUNCTION statement to indicate that one of the input parameters is a transition table), the function signature cannot be used to identify the function. Instead, identify the function with its function name, if unique, or with its specific name.

FUNCTION *function-name*

Identifies the function by its name. The *function-name* must identify exactly one function. The function can have any number of parameters defined for it. If there is more than one function of the specified name in the specified or implicit schema, an error is returned.

An * can be specified for a qualified or unqualified *function-name*. An * (or *schema-name*.*) indicates that the privilege is granted on all the functions in the schema including those that do not currently exist. SYSADM authority is required if an * (or *schema-name*.*) is specified. Specifying an * does not affect any EXECUTE privileges that are already granted on a function.

FUNCTION *function-name (parameter-type,...)*

Identifies the function by its function signature, which uniquely identifies the function. The *function-name (parameter-type, ...)* must identify a function with the specified function signature. The specified parameters must match the data types in the corresponding position that were specified when the function was created. The number of data types, and the logical concatenation of the data types is used to identify the specific function instance on which the privilege is to be granted. Synonyms for data types are considered a match.

If the function was defined with a table parameter (the LIKE TABLE *name* AS LOCATOR clause was specified in the CREATE FUNCTION statement to indicate that one of the input parameters is a transition table), the function signature cannot be used to uniquely identify the function. Instead, use one of the other syntax variations to identify the function with its function name, if unique, or its specific name.

If *function-name ()* is specified, the function identified must have zero parameters.

function-name

Identifies the name of the function. If you do not explicitly qualify the function name with a schema name, the function name is implicitly qualified with a schema name as described in the preceding description for FUNCTION *function-name*.

(parameter-type,...)

Identifies the parameters of the function.

If an unqualified distinct type name is specified, DB2 searches the SQL path to resolve the schema name for the distinct type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parentheses indicate that the database manager ignores the attribute when determining whether the data types match. For

example, DEC() will be considered a match for a parameter of a function defined with a data type of DEC(7,2). Similarly, DECFLOAT() will be considered a match for DECFLOAT(16) or DECFLOAT(34). However, FLOAT cannot be specified with empty parenthesis because its parameter value indicates a specific data type (REAL or DOUBLE).

- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

For data types with a subtype or encoding scheme attribute, specifying the FOR *subtype* DATA clause or the CCSID clause is optional. Omission of either clause indicates that DB2 ignores the attribute when determining whether the data types match. If you specify either clause, it must match the value that was implicitly or explicitly specified in the CREATE FUNCTION statement.

AS LOCATOR
Specifies that the function is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or a distinct type based on a LOB.

SPECIFIC FUNCTION *specific-name*
Identifies the function by its specific name. The *specific-name* must identify a specific function that exists at the current server.

PROCEDURE *procedure-name*
Identifies a stored procedure that is defined at the current server. The name, including the implicit or explicit schema name, must identify a stored procedure that exists at the current server.

An * can be specified for a qualified or unqualified *procedure-name*. An * (or *schema-name*.*) indicates that the privilege is granted on all the stored procedures in the schema including those that do not currently exist. Specifying an * does not affect any EXECUTE privileges that are already granted on a stored procedure.

TO Refer to "GRANT" on page 1695 for a description of the TO clause.

WITH GRANT OPTION
Refer to "GRANT" on page 1695 for a description of the WITH GRANT OPTION clause.

Examples

Example 1: Grant the EXECUTE privilege on function CALC_SALARY to user JONES. Assume that there is only one function in the schema with function name CALC_SALARY.

```
GRANT EXECUTE ON FUNCTION CALC_SALARY TO JONES;
```

Example 2: Grant the EXECUTE privilege on procedure VACATION_ACCR to all users at the current server.

```
GRANT EXECUTE ON PROCEDURE VACATION_ACCR TO PUBLIC;
```

Example 3: Grant the EXECUTE privilege on function DEPT_TOTALS to the administrative assistant and give the assistant the ability to grant the EXECUTE privilege on this function to others. The function has the specific name DEPT85_TOT. Assume that the schema has more than one function that is named DEPT_TOTALS.

```
GRANT EXECUTE ON SPECIFIC FUNCTION DEPT85_TOT TO ADMIN_A
      WITH GRANT OPTION;
```

Example 4: Grant the EXECUTE privilege on function NEW_DEPT_HIRES to HR (Human Resources). The function has two input parameters with data types of INTEGER and CHAR(10), respectively. Assume that the schema has more than one function that is named NEW_DEPT_HIRES.

```
GRANT EXECUTE ON FUNCTION NEW_DEPT_HIRES (INTEGER, CHAR(10))
      TO HR;
```

You can also code the CHAR(10) data type as CHAR().

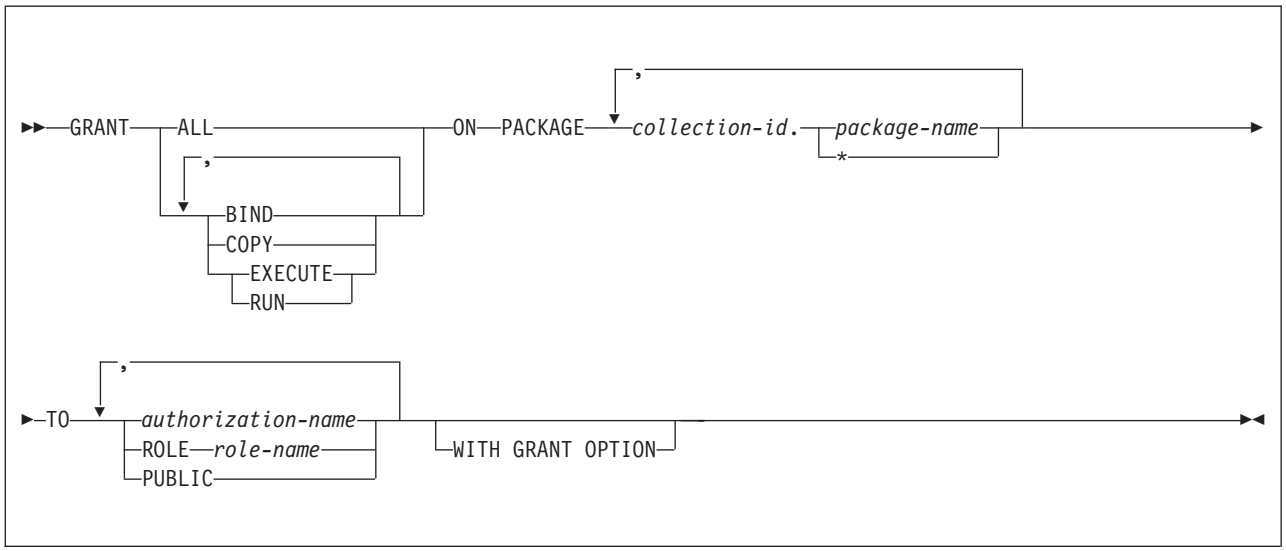
Example 5: Grant the EXECUTE privilege on function FIND_EMPDEPT to role ROLE1:

```
GRANT EXECUTE ON FUNCTION FIND_EMPDEPT TO ROLE ROLE1;
```

GRANT (package privileges)

This form of the GRANT statement grants privileges on packages.

Syntax



Description

BIND
Grants the privilege to use the BIND and REBIND subcommands for the designated packages.

The BIND package privilege can also be used to allow a user to add a new version of an existing package. For details on the authorization required to create new packages and new versions of existing packages, see "Notes" on page 1709.

COPY
Grants the privilege to use the COPY option of the BIND subcommand for the designated packages.

EXECUTE
Grants the privilege to run application programs that use the designated packages and to specify the packages following PKLIST for the BIND PLAN and REBIND PLAN commands. RUN is an alternate name for the same privilege.

ALL
Grants all package privileges for which you have GRANT authority for the packages named in the ON clause.

ON PACKAGE *collection-id.package-name,...*
Identifies packages for which you are granting privileges. The granting of a package privilege applies to all versions of a package. The list can simultaneously contain items of the following two forms:

- *collection-id.package-name* explicitly identifies a single package. The name must identify a package that exists at the current server.
- *collection-id.** applies to every package in the indicated collection. This includes packages that currently exist and future packages. The grant applies to a collection at the current server, but the *collection-id* does not have to identify a collection that exists when the grant is made.

To grant a privilege in this form requires PACKADM with the WITH GRANT OPTION over the collection or all collections, SYSADM, or SYSCTRL authority. Because of this fact, WITH GRANT OPTION, if included in the statement, is ignored for grants of this form, but not for grants for specific packages.

TO Refer to "GRANT" on page 1695 for a description of the TO clause.

WITH GRANT OPTION
Refer to "GRANT" on page 1695 for a description of the WITH GRANT OPTION clause.

Notes

The authorization required to add a new package or a new version of an existing package depends on the value of field BIND NEW PACKAGE on installation panel DSNTIPP. The default value is BINDADD.

If the value of BIND NEW PACKAGE is BINDADD, the owner must have one of the following to add a new package or a new version of an existing package to a collection:

- The BINDADD system privilege and either the CREATE IN privilege or PACKADM authority for the collection or for all collections
- SYSADM or SYSCTRL authority

If the value of BIND NEW PACKAGE is BIND, the owner must have one of the following to add a new package or a new version of an existing package to a collection:

- The BINDADD system privilege and either the CREATE IN privilege or PACKADM authority for the collection or for all collections
- SYSADM or SYSCTRL authority
- PACKADM authority for the collection or for all collections
- Users with the BIND package privilege can also add a new version of an existing package

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports specifying PROGRAM as a synonym for PACKAGE.

Examples

Example 1: Grant the privilege to copy all packages in collection DSN8CC61 to LEWIS.

```
GRANT COPY ON PACKAGE DSN8CC61.* TO LEWIS;
```

Example 2: You have the BIND privilege with GRANT authority over the package CLCT1.PKG1. You have the EXECUTE privilege with GRANT authority over the package CLCT2.PKG2. You have no other privileges with GRANT authority over any package in the collections CLCT1 AND CLCT2. Hence, the following statement, when executed by you, grants LEWIS the BIND privilege on CLCT1.PKG1 and the EXECUTE privilege on CLCT2.PKG2, and makes no other grant. The privileges granted include no GRANT authority.

```
GRANT ALL ON PACKAGE CLCT1.PKG1, CLCT2.PKG2 TO JONES;
```

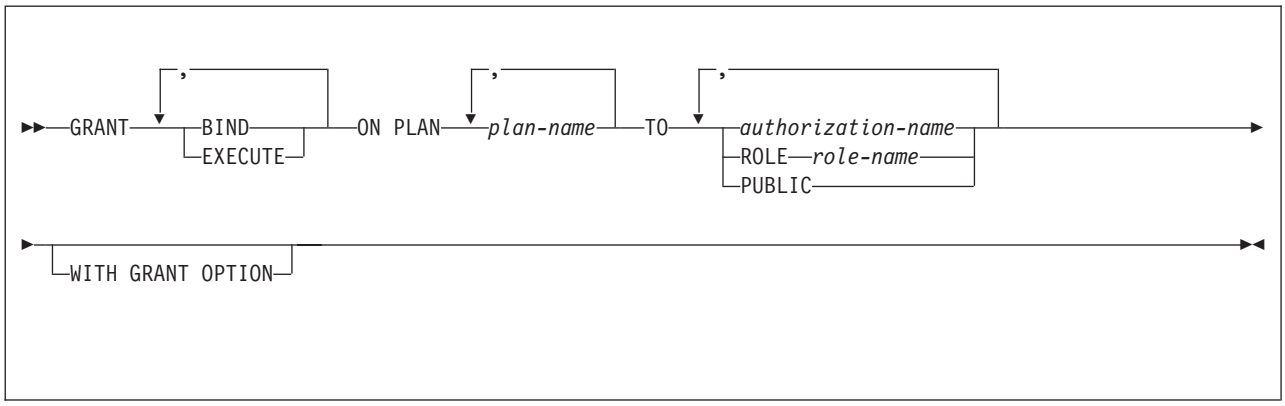
Example 3: Grant the privileges to run all packages in collection DSN9CC13 to role ROLE1:

```
GRANT EXECUTE ON PACKAGE DSN9CC13.* TO ROLE ROLE1;
```

GRANT (plan privileges)

This form of the GRANT statement grants privileges on plans.

Syntax



Description

BIND
Grants the privilege to use the BIND, REBIND, and FREE subcommands for the identified plans. (The authority to create new plans using BIND ADD is a system privilege.)

EXECUTE
Grants the privilege to run programs that use the identified plans.

ON PLAN *plan-name,...*
Identifies the application plans on which the privileges are granted. For each identified plan, you must have all specified privileges with the GRANT option.

TO Refer to "GRANT" on page 1695 for a description of the TO clause.

WITH GRANT OPTION
Refer to "GRANT" on page 1695 for a description of the WITH GRANT OPTION clause.

Examples

Example 1: Grant the privilege to bind plan DSN8IP11 to user JONES.

```
GRANT BIND ON PLAN DSN8IP11 TO JONES;
```

Example 2: Grant privileges to bind and execute plan DSN8CP11 to all users at the current server.

```
GRANT BIND,EXECUTE ON PLAN DSN8CP11 TO PUBLIC;
```

Example 3: Grant the privilege to execute plan DSN8CP11 to users ADAMSON and BROWN with grant option.

```
GRANT EXECUTE ON PLAN DSN8CP11 TO ADAMSON,BROWN WITH GRANT OPTION;
```

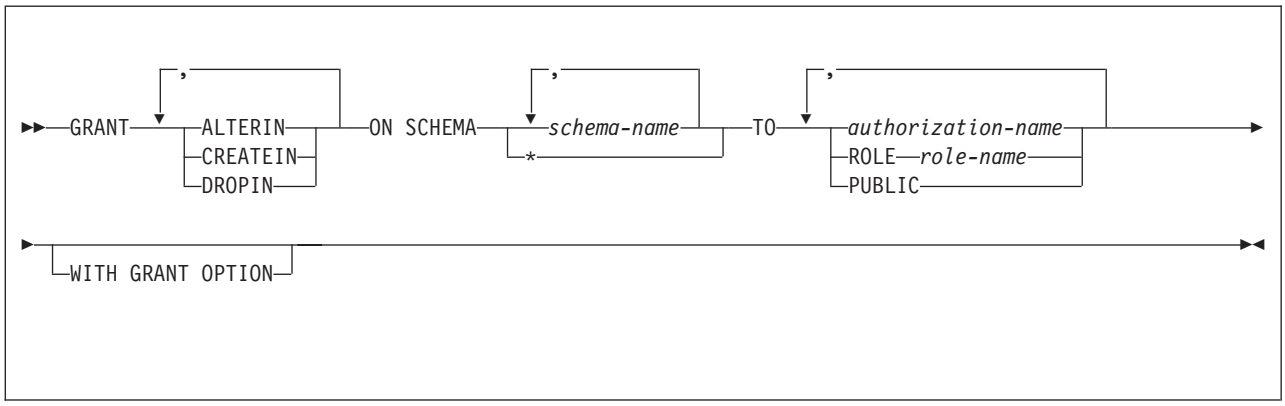
Example 4: Grant the privileges to bind the DSN91PLN plan to role ROLE1:

```
GRANT BIND ON PLAN DSN91PLN TO ROLE ROLE1;
```

GRANT (schema privileges)

This form of the GRANT statement grants privileges on schemas.

Syntax



Description

ALTERIN
Grants the privilege to alter stored procedures and user-defined functions, or specify a comment for distinct types, cast functions that are generated for distinct types, sequences, stored procedures, triggers, and user-defined functions in the designated schemas.

CREATEIN
Grants the privilege to create distinct types, sequences, stored procedures, triggers, and user-defined functions in the designated schemas.

DROPIN
Grants the privilege to drop distinct types, sequences, stored procedures, triggers, and user-defined functions in the designated schemas.

SCHEMA *schema-name*
Identifies the schemas on which the privilege is granted. The schemas do not need to exist when the privilege is granted.

SCHEMA *
Indicates that the specified privilege is granted on all schemas including those that do not currently exist.

TO Refer to "GRANT" on page 1695 for a description of the TO clause.

WITH GRANT OPTION
Refer to "GRANT" on page 1695 for a description of the WITH GRANT OPTION clause.

Notes

Grant on SYSPUBLIC:
Privileges can be granted on the reserved schema SYSPUBLIC. Granting CREATEIN privilege allows the user to create a public alias and granting DROPIN privilege allows the user to drop any public alias.

Examples

Example 1: Grant the CREATEIN privilege on schema T_SCORES to user JONES.

```
GRANT CREATEIN ON SCHEMA T_SCORES TO JONES;
```

Example 2: Grant the CREATEIN privilege on schema VAC to all users at the current server.

```
GRANT CREATEIN ON SCHEMA VAC TO PUBLIC;
```

Example 3: Grant the ALTERIN privilege on schema DEPT to the administrative assistant and give the grantee the ability to grant ALTERIN privileges on this schema to others.

```
GRANT ALTERIN ON SCHEMA DEPT TO ADMIN_A
      WITH GRANT OPTION;
```

Example 4: Grant the CREATEIN, ALTERIN, and DROPIN privileges on schemas NEW_HIRE, PROMO, and RESIGN to HR (Human Resources).

```
GRANT CREATEIN, ALTERIN, DROPIN ON SCHEMA NEW_HIRE, PROMO, RESIGN TO HR;
```

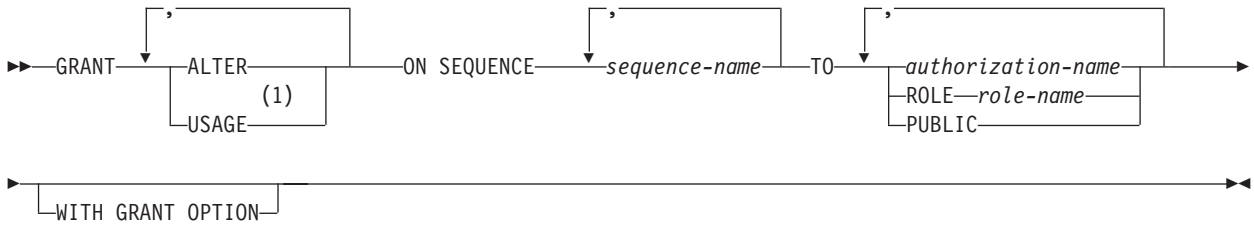
Example 5: Grant the ALTERIN privileges on the EMPLOYEE schema to role ROLE1:

```
GRANT ALTERIN ON SCHEMA EMPLOYEE TO ROLE ROLE1;
```

GRANT (sequence privileges)

This form of the GRANT statement grants privileges on a user-defined sequence.

Syntax



Notes:

1 The keyword SELECT is an alternative keyword for USAGE.

Description

ALTER
Grants the privilege to alter a sequence or record a comment on a sequence.

USAGE
Grants the USAGE privilege to use a sequence. This privilege is needed when the NEXT VALUE or PREVIOUS VALUE expression is invoked for a sequence name.

SEQUENCE *sequence-name*
Identifies the sequence. The name, including the implicit or explicit schema qualifier, must uniquely identify an existing sequence at the current server. If no sequence by this name exists in the explicitly or implicitly specified schema, an error occurs. *sequence-name* must not be the name of an internal sequence object that is used by DB2.

TO Refer to "GRANT" on page 1695 for a description of the TO clause.

WITH GRANT OPTION
Refer to "GRANT" on page 1695 for a description of the WITH GRANT OPTION clause.

Examples

Example 1: Grant USAGE privilege on sequence MYNUM to user JONES.

```
GRANT USAGE
  ON SEQUENCE MYNUM
  TO JONES;
```

Example 2: Grant USAGE privileges on sequence ORDER_SEQ to role ROLE1:

```
GRANT USAGE ON SEQUENCE ORDER_SEQ TO ROLE ROLE1;
```

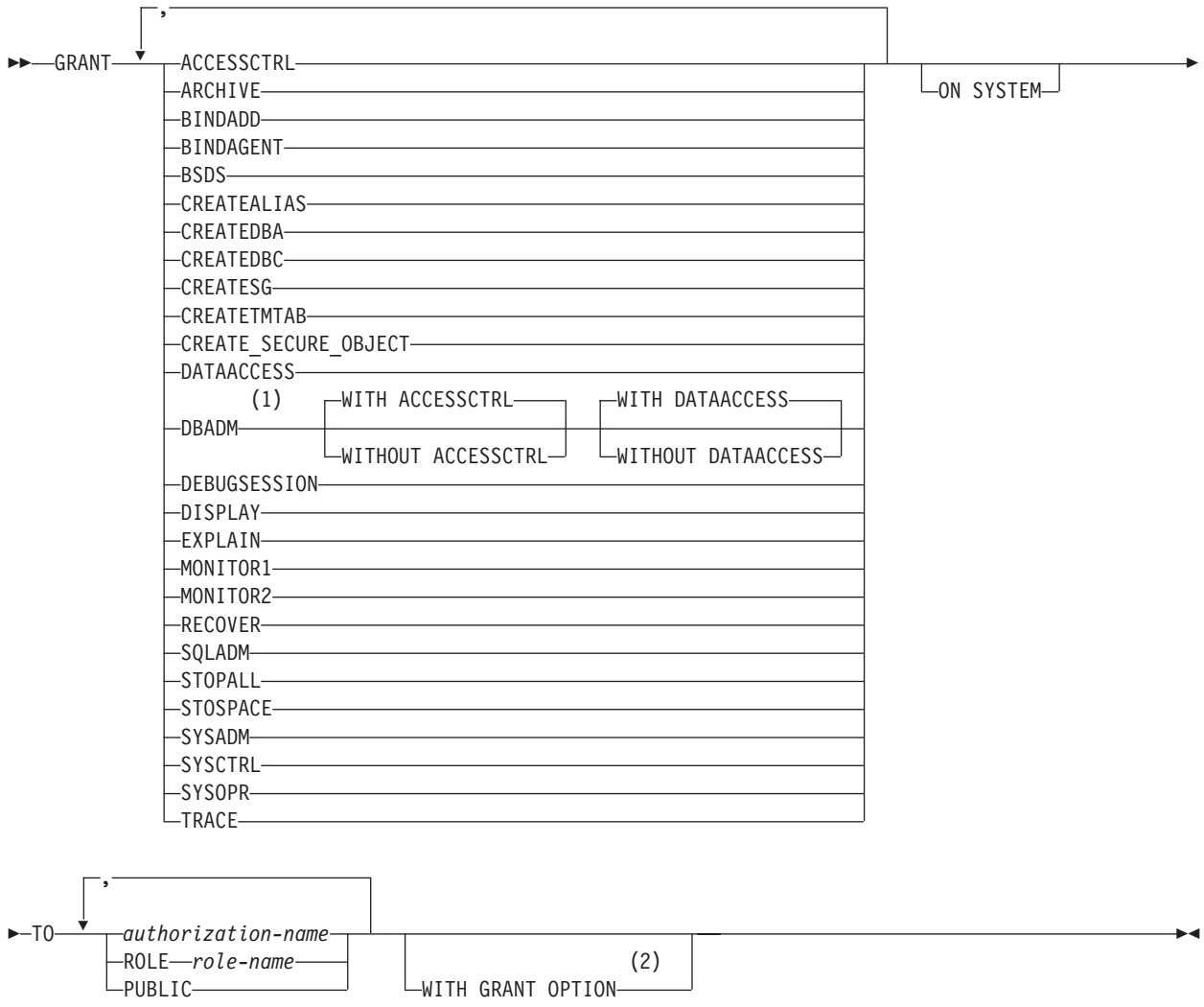
GRANT (system privileges)

This form of the GRANT statement grants system privileges.

Authorization

To grant the CREATE_SECURE_OBJECT system privilege, the privileges that are held by the authorization ID of the statement must include SECADM authority.

Syntax



Notes:

1 The ACCESSCTRL and DATAACCESS clauses can be specified in any order.

2 The WITH GRANT OPTION can be specified but is ignored for DBADM, DATAACCESS, and ACCESSCTRL.

Description

ACCESSCTRL
Grants the ACCESSCTRL authority. ACCESSCTRL allows the user to grant all authorities and privileges, except system DBADM, DATAACCESS, ACCESSCTRL, and privileges on security related objects.

A warning is issued if the WITH GRANT OPTION is specified when granting this authority.

ACCESSCTRL cannot be granted to PUBLIC.

ARCHIVE
Grants the privilege to use the ARCHIVE LOG and SET LOG commands.

BINDADD
Grants the privilege to create plans and packages by using the BIND subcommand with the ADD option.

BINDAGENT
Grants the privilege to issue the BIND, FREE PACKAGE, or REBIND subcommands for plans and packages and the DROP PACKAGE statement on behalf of the grantor. The privilege also allows the holder to copy and replace plans and packages on behalf of the grantor.

A warning is issued if WITH GRANT OPTION is specified when granting this privilege.

BSDS
Grants the privilege to issue the RECOVER BSDS command.

CREATEALIAS
Grants the privilege to use the CREATE ALIAS statement.

CREATEDBA
Grants the privilege to issue the CREATE DATABASE statement and acquire DBADM authority over those databases.

CREATEDBC
Grants the privilege to issue the CREATE DATABASE statement and acquire DBCTRL authority over those databases.

CREATESG
Grants the privilege to create new storage groups.

CREATETMTAB
Grants the privilege to use the CREATE GLOBAL TEMPORARY TABLE statement.

CREATE_SECURE_OBJECT
Grants the privilege to create a secure object.

DATAACCESS
Grants the DATAACCESS authority. DATAACCESS allows the user to access data in all user tables, views, and materialized query tables in a DB2 subsystem and allows the user to execute plans, packages, functions, and procedures.

A warning is issued if the WITH GRANT OPTION is specified when granting this authority.

DATAACCESS cannot be granted to PUBLIC.

DBADM
Grants the DBADM authority. DBADM allows the user to manage all objects in the DB2 subsystem, except security objects.

A warning is issued if the WITH GRANT OPTION is specified when granting this authority.

DBADM cannot be granted to PUBLIC.

WITH ACCESSCTRL
Specifies that the ACCESSCTRL authority is granted along with the system DBADM authority. ACCESSCTRL allows system DBADM to grant all authorities and privileges, except system DBADM, DATAACCESS, ACCESSCTRL authorities and privileges on security related objects. ACCESSCTRL can be used to REVOKE privileges using the BY clause.

WITH ACCESSCTRL is the default.

WITHOUT ACCESSCTRL
Specifies that system DBADM authority is not granted the ACCESSCTRL authority.

WITH DATAACCESS
Specifies that the DATAACCESS authority is granted along with the system DBADM authority. DATAACCESS allows the system DBADM to access data in all user tables, views, and materialized query tables in a DB2 subsystem and allows the user to execute plans, packages, functions, and procedures.

WITH DATAACCESS is the default.

WITHOUT DATAACCESS
Specifies that system DBADM authority is not granted the DATAACCESS authority.

DISPLAY
Grants the privilege to use the following commands:

- The DISPLAY ARCHIVE command for archive log information
- The DISPLAY BUFFERPOOL command for the status of buffer pools
- The DISPLAY DATABASE command for the status of all databases
- The DISPLAY LOCATION command for statistics about threads with a distributed relationship
- The DISPLAY LOG command for log information, including the status of the offload task
- The DISPLAY THREAD command for information on active threads within DB2
- The DISPLAY TRACE command for a list of active traces

DEBUGSESSION
Grants the privilege to attach a debug client to the current application process connection, which enables client application debugging of native SQL or Java procedures that are executed within the session.

EXPLAIN
Grants the privilege to issue the following without requiring the privileges needed to execute the statement:

- EXPLAIN statement with the options:
  - PLAN
  - ALL

- PREPARE statement
- DESCRIBE TABLE statement
- Explain dynamic SQL statements that execute under the special register CURRENT EXPLAIN MODE, when CURRENT EXPLAIN MODE = EXPLAIN
- BIND options: EXPLAIN(ONLY) and SQLERROR(CHECK)

  EXPLAIN(ONLY) allows to explain the statements.

  SQLERROR(CHECK) performs all syntax and semantic checks on the SQL statements that are being bound.

MONITOR1
Grants the privilege to obtain IFC data classified as serviceability data, statistics, accounting, and other performance data that does not contain potentially secure data.

MONITOR2
Grants the privilege to obtain IFC data classified as containing potentially sensitive data such as SQL statement text and audit data. Users with MONITOR2 privileges have MONITOR1 privileges.

RECOVER
Grants the privilege to issue the RECOVER INDOUBT command.

SQLADM
Grants the authority to perform the following actions without requiring any additional privileges:

- DESCRIBE TABLE statement
- EXPLAIN statement with the following options:
  - PLAN
  - ALL
  - STMTCACHE ALL
  - STMTID
  - STMTTOKEN
  - MONITORED STMTS
- PREPARE statement
- Explain dynamic SQL statements that execute under the special register CURRENT EXPLAIN MODE, when CURRENT EXPLAIN MODE = EXPLAIN
- BIND options: EXPLAIN(ONLY) and SQLERROR(CHECK)

  EXPLAIN(ONLY) allows to explain the statements.

  SQLERROR(CHECK) performs all syntax and semantic checks on the SQL statements that are being bound.
- START command
- STOP command
- DISPLAY PROFILE command
- Execute the RUNSTATS utility and the MODIFY STATISTICS utility in any database.
- MONITOR2 privilege to obtain IFC data classified as containing potentially sensitive data, such as SQL statement text and audit data, as well as IFC data classified as serviceability data, statistics, accounting, and other performance data.

STOPALL
Grants the privilege to issue the STOP DB2 command.

STOSPACE
Grants the privilege to use the STOSPACE utility.

SYSADM
Grants all DB2 privileges except for a few reserved for installation SYSADM authority. The privileges the user possesses are all grantable, including the SYSADM authority itself. The privileges the user lacks restrict what the user can do with the directory and the catalog. Using WITH GRANT OPTION when granting SYSADM is redundant but valid. For more on SYSADM and installation SYSADM authority, see *DB2 Administration Guide*.

SYSCTRL
Grants the system control authority, which allows the user to have most of the privileges of a system administrator but excludes the privileges to read or change user data. Using WITH GRANT OPTION when granting SYSCTRL is redundant but valid. For more information on SYSCTRL authority, see *DB2 Administration Guide*.

SYSOPR
Grants the privilege to have system operator authority.

TRACE
Grants the privilege to issue the MODIFY TRACE, START TRACE, and STOP TRACE commands.

TO Refer to "GRANT" on page 1695 for a description of the TO clause.

WITH GRANT OPTION
If you grant the SYSADM or SYSCTRL system privilege, WITH GRANT OPTION is valid but unnecessary. It is unnecessary because whoever is granted SYSADM or SYSCTRL has that authority and all the privileges it implies, with the GRANT option.

Examples

Example 1: Grant DISPLAY privileges to user LUTZ.

```
GRANT DISPLAY
  TO LUTZ;
```

Example 2: Grant BSDS and RECOVER privileges to users PARKER and SETRIGHT, with the WITH GRANT OPTION.

```
GRANT BSDS,RECOVER
  TO PARKER,SETRIGHT
  WITH GRANT OPTION;
```

Example 3: Grant TRACE privileges to all local users.

```
GRANT TRACE
  TO PUBLIC;
```

Example 4: Grant ARCHIVE privileges to role ROLE1:

```
GRANT ARCHIVE TO ROLE ROLE1;
```

Example 5: SECADM Linda grants the privilege to Steve to create a secure object:

```
 GRANT CREATE_SECURE_OBJECT
TO STEVE;
```

Example 6: Grant system DBADM with ACCESSCTRL and with DATAACCESS to role, ADMINROLE and authid, SALLY. Since GRANT system DBADM also grants ACCESSCTRL and DATAACCESS by default, WITH ACCESSCTRL and WITH DATAACCESS clauses need not be specified explicitly.

```
  GRANT DBADM ON SYSTEM
 TO ROLE ADMINROLE;
GRANT DBADM, ACCESSCTRL, DATAACCESS
 ON SYSTEM
 TO SALLY;
```

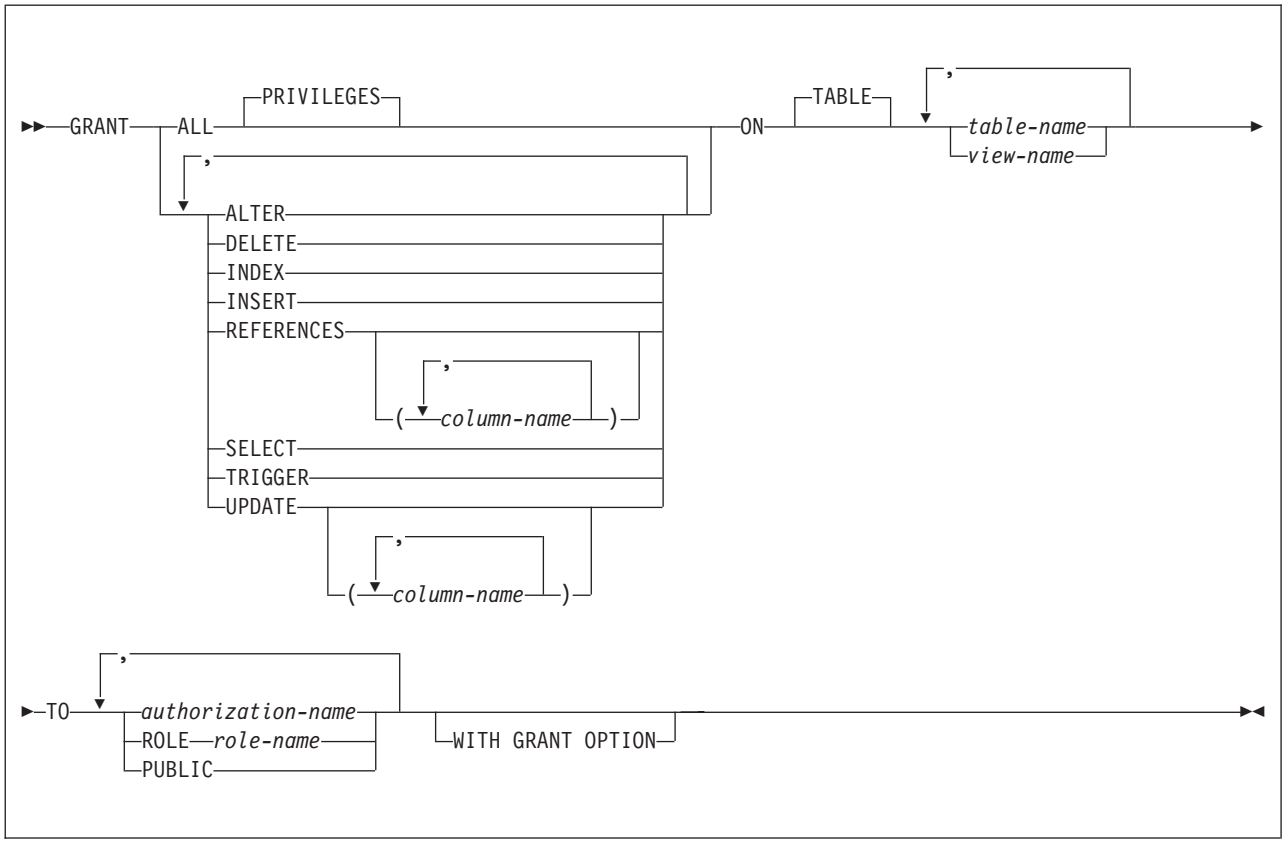
Example 7: Grant system DBADM without ACCESSCTRL and without DATAACCESS to John. The WITHOUT ACCESSCTRL and WITHOUT DATAACCESS clauses need to be specified explicitly.

```
  GRANT DBADM WITHOUT ACCESSCTRL
 WITHOUT DATAACCESS
 ON SYSTEM
 TO JOHN;
```

GRANT (table or view privileges)

This form of the GRANT statement grants privileges on tables and views.

Syntax



Description

ALL or ALL PRIVILEGES

Grants all table or view privileges for which you have GRANT authority, for the tables and views named in the ON clause.

If you do not use ALL, you must use one or more of the keywords in the following list. For each keyword that you use, you must have GRANT authority for that privilege on every table or view identified in the ON clause.

ALTER

Grants the privilege to alter the specified table or create a trigger on the specified table. ALTER cannot be used if the statement identifies an auxiliary table or a view.

DELETE

Grants the privilege to delete rows in the specified table or view. DELETE cannot be granted on an auxiliary table.

INDEX

Grants the privilege to create an index on the specified table. INDEX cannot be granted on a view.

INSERT
Grants the privilege to insert rows into the specified table or view. INSERT cannot be granted on an auxiliary table.

REFERENCES
Grants the privilege to add a referential constraint in which the specified table is a parent. If a list of column names is not specified or if REFERENCES is granted via the specification of ALL PRIVILEGES, the grantee can define referential constraints using all columns of the table as a parent key, even those added later via the ALTER TABLE statement. This privilege cannot be granted on a view or auxiliary table.

REFERENCES(*column-name***,...)**
Grants the privilege to add or drop a referential constraint in which the specified table is a parent using only those columns that are specified in the column list as a parent key. Each *column-name* must be an unqualified name that identifies a column of the table identified in the ON clause. This privilege cannot be granted on a view or auxiliary table.

SELECT
Grants the privilege to create a view or read data from the specified table or view. SELECT cannot be granted on an auxiliary table.

TRIGGER
Grants the privilege to create a trigger on the specified table. TRIGGER cannot be granted on an auxiliary table or a view.

UPDATE
Grants the privilege to update rows in the specified table or view. UPDATE cannot be granted on an auxiliary table.

UPDATE(*column-name***,...)**
Grants the privilege to update only the columns named. Each *column-name* must be the unqualified name of a column of every table or view identified in the ON clause. Each *column-name* must not identify a column of an auxiliary table.

ON *table-name* **or** *view-name*
Specifies the tables or views on which you are granting the privileges. The list can be a list of table names or view names, or a combination of the two. A declared temporary table and a table that is implicitly created for an XML column must not be identified.

If you use GRANT ALL, then for each named table or view, the privilege set (described in ""Authorization" on page 1695" in "GRANT" on page 1695) must include at least one privilege with the GRANT option.

TO Refer to "GRANT" on page 1695 for a description of the TO clause.

WITH GRANT OPTION
Refer to "GRANT" on page 1695 for a description of the WITH GRANT OPTION clause.

Notes

The REFERENCES privilege does not replace the ALTER privilege. It was added to conform to the SQL standard. To define a foreign key that references a parent table, you must have either the REFERENCES or the ALTER privilege, or both.

For a created temporary table, only ALL or ALL PRIVILEGES can be granted. Specific table privileges cannot be granted. In addition, only the ALTER, DELETE, INSERT, and SELECT privileges apply to a created temporary table.

For a view of a created temporary table, either ALL or the specific UPDATE, DELETE, INSERT and SELECT privileges can be granted. When ALL is specified only the UPDATE, DELETE, INSERT, and SELECT privileges apply to a view on created temporary table. However, the UPDATE operation of the view is not allowed.

To grant table privileges on a created temporary table, the privilege set must include one of the following:

- SYSADM
- DBADM on DSNDB06
- Ownership of the created temporary table

To grant table privileges on a view of a created temporary table, the privilege set must include one of the following:

- SYSADM
- ownership of the created temporary table

For a declared temporary table, no privileges can be granted. When a declared temporary table is defined, PUBLIC implicitly receives all table privileges (without GRANT authority) for the table. These privileges are not recorded in the DB2 catalog, and they cannot be revoked.

For an auxiliary table, only the INDEX privilege can be granted. DELETE, INSERT, SELECT, and UPDATE privileges on the base table that is associated with the auxiliary table extend to the auxiliary table.

- ALTER
- INDEX
- REFERENCES
- TRIGGER

Examples

Example 1: Grant SELECT privileges on table DSN8B10.EMP to user PULASKI.

```
GRANT SELECT ON DSN8B10.EMP TO PULASKI;
```

Example 2: Grant UPDATE privileges on columns EMPNO and WORKDEPT in table DSN8B10.EMP to all users at the current server.

```
GRANT UPDATE (EMPNO,WORKDEPT) ON TABLE DSN8B10.EMP TO PUBLIC;
```

Example 3: Grant all privileges on table DSN8B10.EMP to users KWAN and THOMPSON, with the WITH GRANT OPTION.

```
GRANT ALL ON TABLE DSN8B10.EMP TO KWAN,THOMPSON WITH GRANT OPTION;
```

Example 4: Grant the SELECT and UPDATE privileges on the table DSN8B10.DEPT to every user in the network.

```
GRANT SELECT, UPDATE ON TABLE DSN8B10.DEPT
  TO PUBLIC;
```

Even with this grant, it is possible that some network users do not have access to the table at all, or to any other object at the subsystem where the table exists. Controlling access to the subsystem involves the communications databases at the subsystems in the network. The tables for the communication databases are described in “DB2 catalog tables” on page 2102. Controlling access is described in *DB2 Administration Guide*.

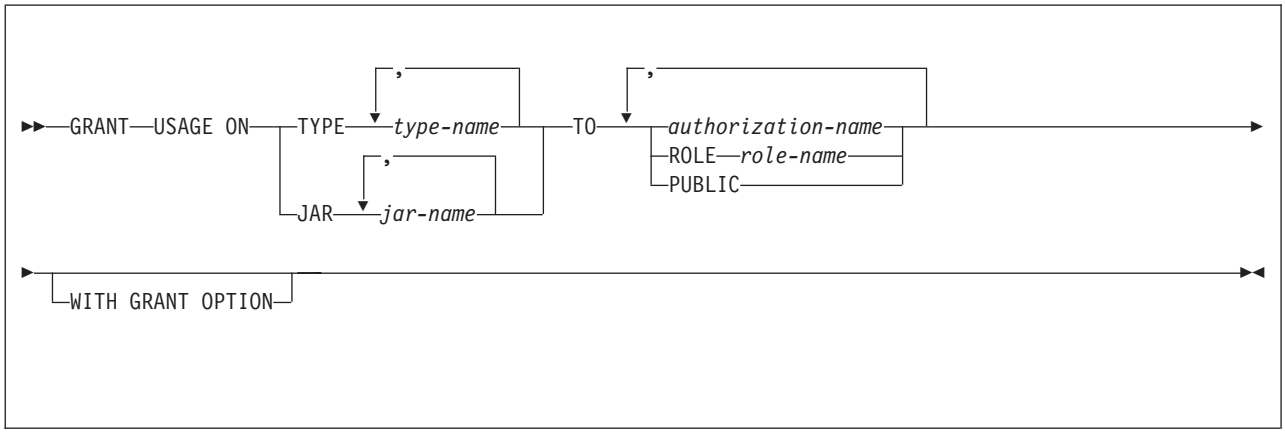
Example 5: Grant ALTER privileges on table DSN9910.EMP to role ROLE1:

```
GRANT ALTER ON TABLE DSN9910.EMP TO ROLE ROLE1;
```

GRANT (type or JAR file privileges)

This form of the GRANT statement grants the privilege to use distinct types, array types, or JAR files.

Syntax



Description

USAGE
Grants the privilege to use the distinct type in tables, functions procedures, or the privilege to use the JAR file.

TYPE *type-name*
Identifies the user-defined type. The name, including the implicit or explicit schema name, must identify a unique user-defined type that exists at the current server.

JAR *jar-name*
Identifies the JAR file. The name, including the implicit or explicit schema name, must identify a unique JAR file that exists at the current server.

TO Refer to "GRANT" on page 1695 for a description of the TO clause.

WITH GRANT OPTION
Refer to "GRANT" on page 1695 for a description of the WITH GRANT OPTION clause.

Notes

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports DATA TYPE or DISTINCT TYPE as a synonym for TYPE.

Examples

Example 1: Grant the USAGE privilege on distinct type SHOE_SIZE to user JONES. This GRANT statement does not give JONES the privilege to execute the cast functions that are associated with the distinct type SHOE_SIZE.

```
GRANT USAGE ON TYPE SHOE_SIZE TO JONES;
```

Example 2: Grant the USAGE privilege on distinct type US_DOLLAR to all users at the current server.

```
GRANT USAGE ON TYPE US_DOLLAR TO PUBLIC;
```

Example 3: Grant the USAGE privilege on distinct type CANADIAN_DOLLAR to the administrative assistant (ADMIN_A), and give this user the ability to grant the USAGE privilege on the distinct type to others. The administrative assistant cannot grant the privilege to execute the cast functions that are associated with the distinct type CANADIAN_DOLLAR because WITH GRANT OPTION does not give the administrative assistant the EXECUTE authority on these cast functions.

```
GRANT USAGE ON TYPE CANADIAN_DOLLAR TO ADMIN_A
      WITH GRANT OPTION;
```

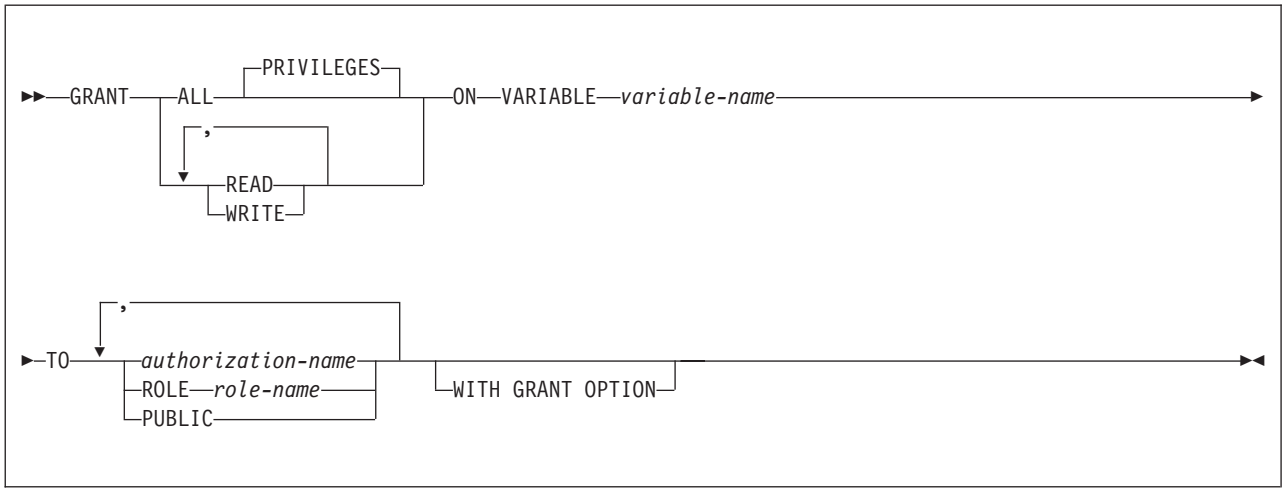
Example 4: Grant the USAGE privilege on the distinct type MILES to role ROLE1 at the current server:

```
GRANT USAGE ON TYPE MILES
    TO ROLE ROLE1;
```

GRANT (variable privileges)

This form of the GRANT statement grants privileges on global variables.

Syntax



Description

ALL PRIVILEGES
Grants both READ and WRITE privileges on the specified global variable.

READ
Grants the privilege to access the content of the specified global variable.

WRITE
Grants the privilege to modify the content of the specified global variable.

ON VARIABLE *variable-name*
Identifies the global variable for which you are granting privileges. *variable-name*, including an implicit or explicit qualifier, must identify a global variable that exists at the current server.

TO Refer to "GRANT" on page 1695 for a description of the TO clause.

WITH GRANT OPTION
Refer to "GRANT" on page 1695 for a description of the WITH GRANT OPTION clause.

Examples

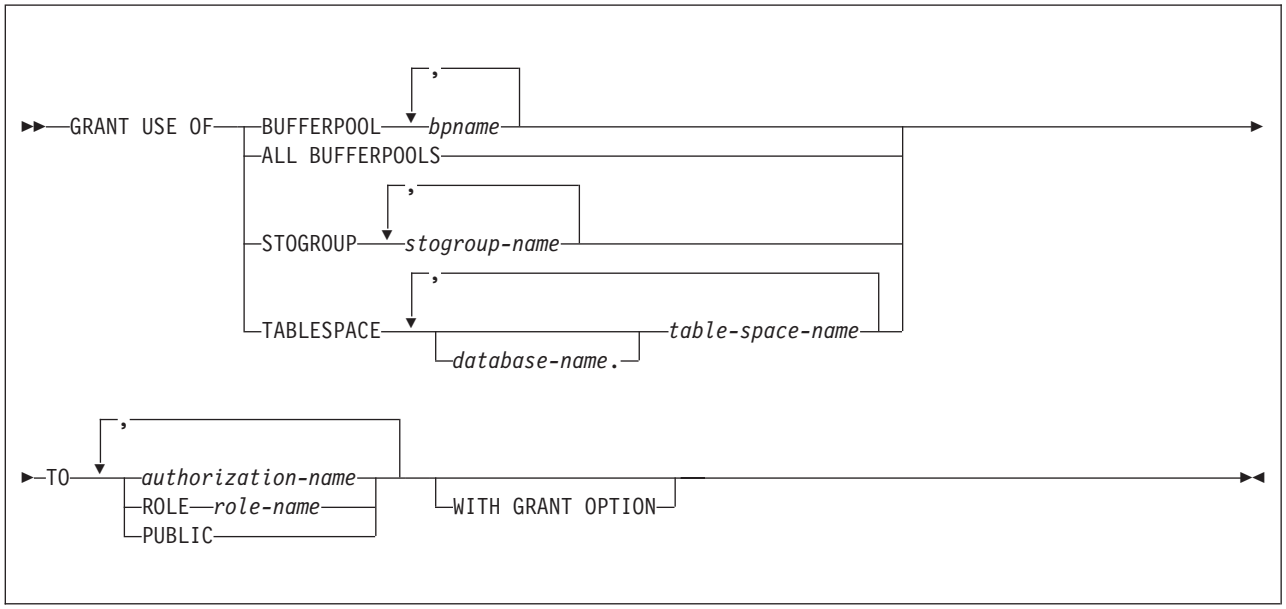
Example 1: Grant the read privilege on the ACCOUNTNO variable on the current server to user Jones:

```
GRANT READ ON VARIABLE ACCOUNTNO TO JONES;
```

GRANT (use privileges)

This form of the GRANT statement grants authority to use particular buffer pools, storage groups, or table spaces.

Syntax



Description

BUFFERPOOL *bpname,...*
Grants the privilege to refer to any of the identified buffer pools in a CREATE INDEX, CREATE TABLESPACE, ALTER INDEX, or ALTER TABLESPACE statement. See "Naming conventions" on page 57 for more details about *bpname*.

ALL BUFFERPOOLS
Grants the privilege to refer to any buffer pool in a CREATE INDEX, CREATE TABLESPACE, ALTER INDEX, or ALTER TABLESPACE statement.

STOGROUP *stogroup-name,...*
Grants the privilege to refer to any of the identified storage groups in a CREATE INDEX, CREATE TABLESPACE, ALTER INDEX, or ALTER TABLESPACE statement.

TABLESPACE *database-name.table-space-name,...*
Grants the privilege to refer to any of the identified table spaces in a CREATE TABLE statement. The default for *database-name* is DSNDB04.

You cannot grant the privilege for table spaces that are for declared temporary tables (table spaces in a work file database). For these table spaces, PUBLIC implicitly has the TABLESPACE privilege (without GRANT authority); this privilege is not recorded in the DB2 catalog, and it cannot be revoked.

TO Refer to "GRANT" on page 1695 for a description of the TO clause.

WITH GRANT OPTION
Refer to "GRANT" on page 1695 for a description of the WITH GRANT OPTION clause.

Notes

You can grant privileges for only one type of object with each statement. Thus, you can grant the use of several table spaces with one statement, but not the use of a table space and a storage group. For each object you identify, you must have the USE privilege with GRANT authority.

Examples

Example 1: Grant authority to use buffer pools BP1 and BP2 to user MARINO.

```
GRANT USE OF BUFFERPOOL BP1,BP2
  TO MARINO;
```

Example 2: Grant to all local users the authority to use table space DSN8S11D in database DSN8D11A.

```
GRANT USE OF TABLESPACE
  DSN8D11A.DSN8S11D
  TO PUBLIC;
```

Example 3: Grant authority to use storage group SG1 to role ROLE1:

```
GRANT USE OF STOGROUP SG1
   TO ROLE ROLE1;
```

HOLD LOCATOR

The HOLD LOCATOR statement allows a LOB locator variable to retain its association with a value beyond a unit of work.

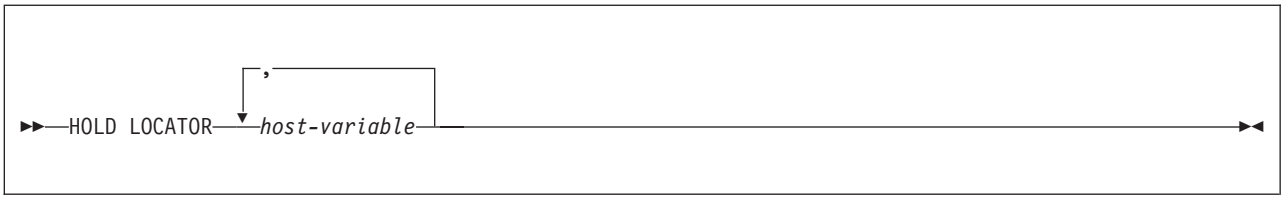
Invocation

This statement can only be embedded in an application program. It cannot be issued interactively. It is an executable statement that can be dynamically prepared. However, the EXECUTE statement with the USING clause must be used to execute the prepared statement. HOLD LOCATOR cannot be used with the EXECUTE IMMEDIATE statement.

Authorization

None required.

Syntax



Description

host-variable, ...
Identifies one or more locator variables that must be declared in accordance with the rules for declaring locator variables. The locator variable type must be a binary large object locator, a character large object locator, or a double-byte character large object locator.

The *host-variable* must currently have a locator assigned to it. That is, a locator must have been assigned during this unit of work (by a FETCH, SELECT INTO, assignment statement, SET *host-variable* statement, or VALUES INTO statement); otherwise, an error is returned.

If more than one locator is specified and an error is returned on one of the locators, it is possible that some locators have been held and others have not been held.

Notes

A host-variable LOB locator variable that has the hold property is freed (has its association between it and its value removed) when:

- The SQL FREE LOCATOR statement is executed for the locator variable.
- The SQL ROLLBACK statement is executed.
- The SQL session is terminated.

Example

Assume that the employee table contains columns RESUME, HISTORY, and PICTURE and that locators have been established in a program to represent the

values represented by the columns. Give the CLOB locator variables LOCRES and LOCHIST, and the BLOB locator variable LOCPIC the hold property.

```
EXEC SQL HOLD LOCATOR :LOCRES, :LOCHIST, :LOCPIC
```

INCLUDE

The INCLUDE statement inserts application code, including declarations and statements, into a source program.

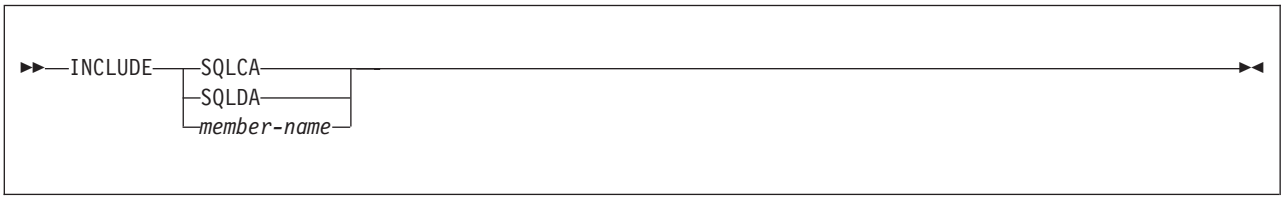
Invocation

This statement can only be embedded in an application program. It is not an executable statement. It must not be specified in Java or REXX.

Authorization

None required.

Syntax



Description

SQLCA
Indicates that the description of an SQL communication area (SQLCA) is to be included. INCLUDE SQLCA must not be specified more than once in the same application program. In COBOL, INCLUDE SQLCA must be specified in the Working-Storage Section or the Linkage Section. INCLUDE SQLCA must not be specified if the program is prepared (either with the DB2 precompiler or coprocessor) with the STDSQL(YES) SQL processing option.

For a description of the SQLCA, see "SQL communication area (SQLCA)" on page 2069.

SQLDA
Indicates that the description of an SQL descriptor area (SQLDA) is to be included. It must not be specified in a Fortran. For a description of the SQLDA, see "SQL descriptor area (SQLDA)" on page 2079.

member-name
Names a member of the partitioned data set to be the library input when your application program is prepared (either with the DB2 precompiler or coprocessor). It must be an SQL identifier.

The member can contain any host language source statements and any SQL statements other than an INCLUDE statement. In COBOL, INCLUDE *member-name* must not be specified in other than the Data Division or the Procedure Division.

Notes

When your application program is prepared (either with the DB2 precompiler or coprocessor), the INCLUDE statement is replaced by source statements. Thus, the INCLUDE statement must be specified at a point in your application program where the resulting source statements are acceptable to the compiler.

The INCLUDE statement cannot refer to source statements that themselves contain INCLUDE statements.

The declarations that are generated by DCLGEN can be used in an application program by specifying the same member in the INCLUDE statement as in the DCLGEN LIBRARY parameter.

Example

Include an SQL communications area in a PL/I program.

```
EXEC SQL INCLUDE SQLCA;
```

INSERT

The INSERT statement inserts rows into a table or view or activates the INSTEAD OF INSERT trigger. The table or view can be at the current server or any DB2 subsystem with which the current server can establish a connection. Inserting a row into a view inserts the row into the table on which the view is based if no INSTEAD OF INSERT trigger is defined on the specified view. If an INSTEAD OF INSERT trigger is defined, the trigger is activated instead of the INSERT statement.

There are three forms of this statement:

- The INSERT via VALUES form is used to insert a single row into the table or view using the values provided or referenced.
- The INSERT via SELECT form is used to insert one or more rows into the table or view using values from other tables, or views, or both.
- The INSERT via FOR *n* ROWS form is used to insert multiple rows into the table or view using values provided or referenced. Although not required, the values can come from *host-variable arrays*. This form of INSERT is supported in SQL procedure applications. However, since host-variable arrays are not supported in SQL procedure applications, the support is limited to insertion of scalar values.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

Authority requirements depend on whether the object identified in the statement is a user-defined table, a catalog table for which inserts are allowed, or a view:

When a user-defined table is identified: The privilege set must include at least one of the following:

- The INSERT privilege on the table
- Ownership of the table
- DBADM authority on the database that contains the table
- SYSADM authority
- DATAACCESS authority

If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

When a catalog table is identified: The privilege set must include at least one of the following:

- DBADM authority on the catalog database
- SYSCTRL authority
- SYSADM authority
- DATAACCESS authority

When a view is identified: The privilege set must include at least one of the following:

- The INSERT privilege on the view
- SYSADM authority

- DATAACCESS authority

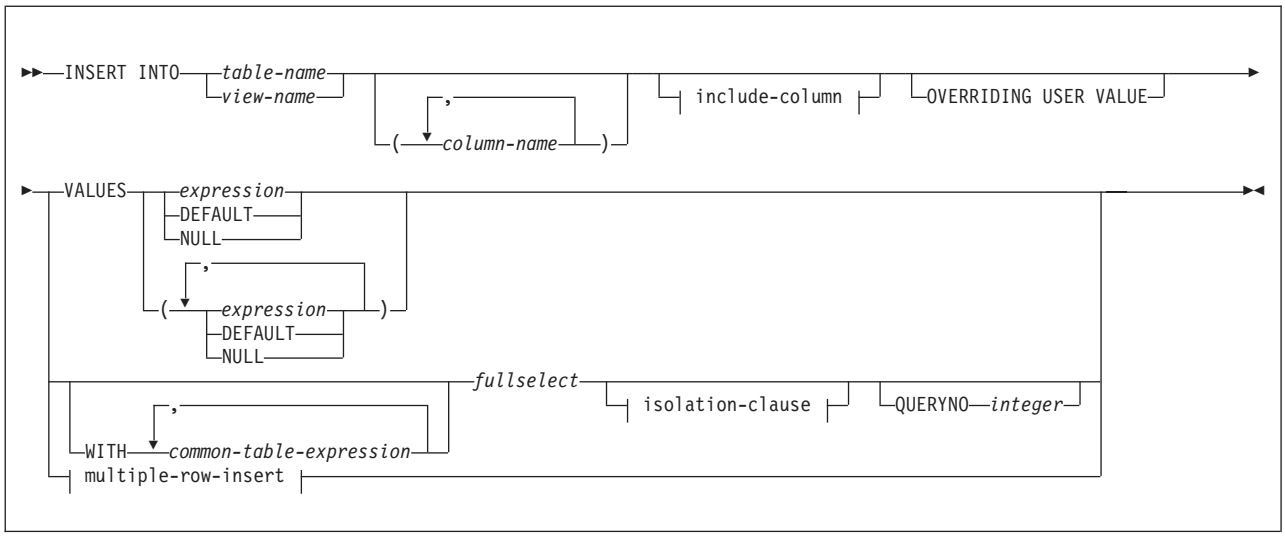
The owner of a view, unlike the owner of a table, might not have INSERT authority on the view (or can have INSERT authority without being able to grant it to others). The nature of the view itself can preclude its use for INSERT. For more information, see the discussion of authority in "CREATE VIEW" on page 1527.

If the INSERT statement is embedded in a SELECT statement, the privilege set must include the SELECT privilege on the table or view.

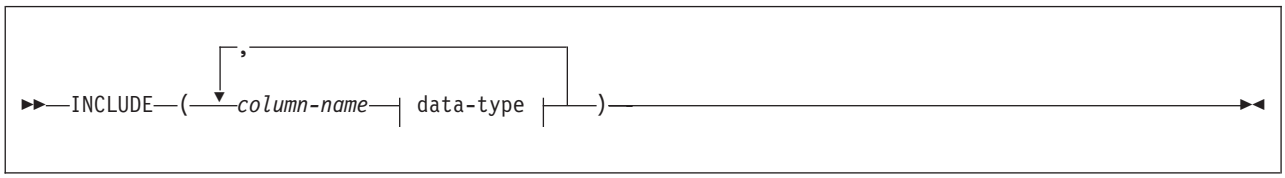
If a fullselect is specified, the privilege set must include authority to execute the fullselect. For more information about the authorization rules, see "Authorization" on page 762.

If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set is determined by the DYNAMICRULES behavior in effect (run, bind, define, or invoke) and is summarized in Table 94 on page 841. (For more information on these behaviors, including a list of the DYNAMICRULES bind option values that determine them, see "Authorization IDs and dynamic SQL" on page 75.)

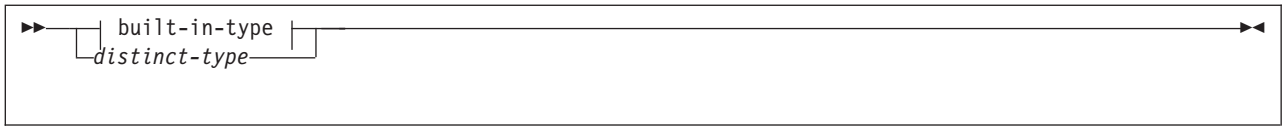
Syntax



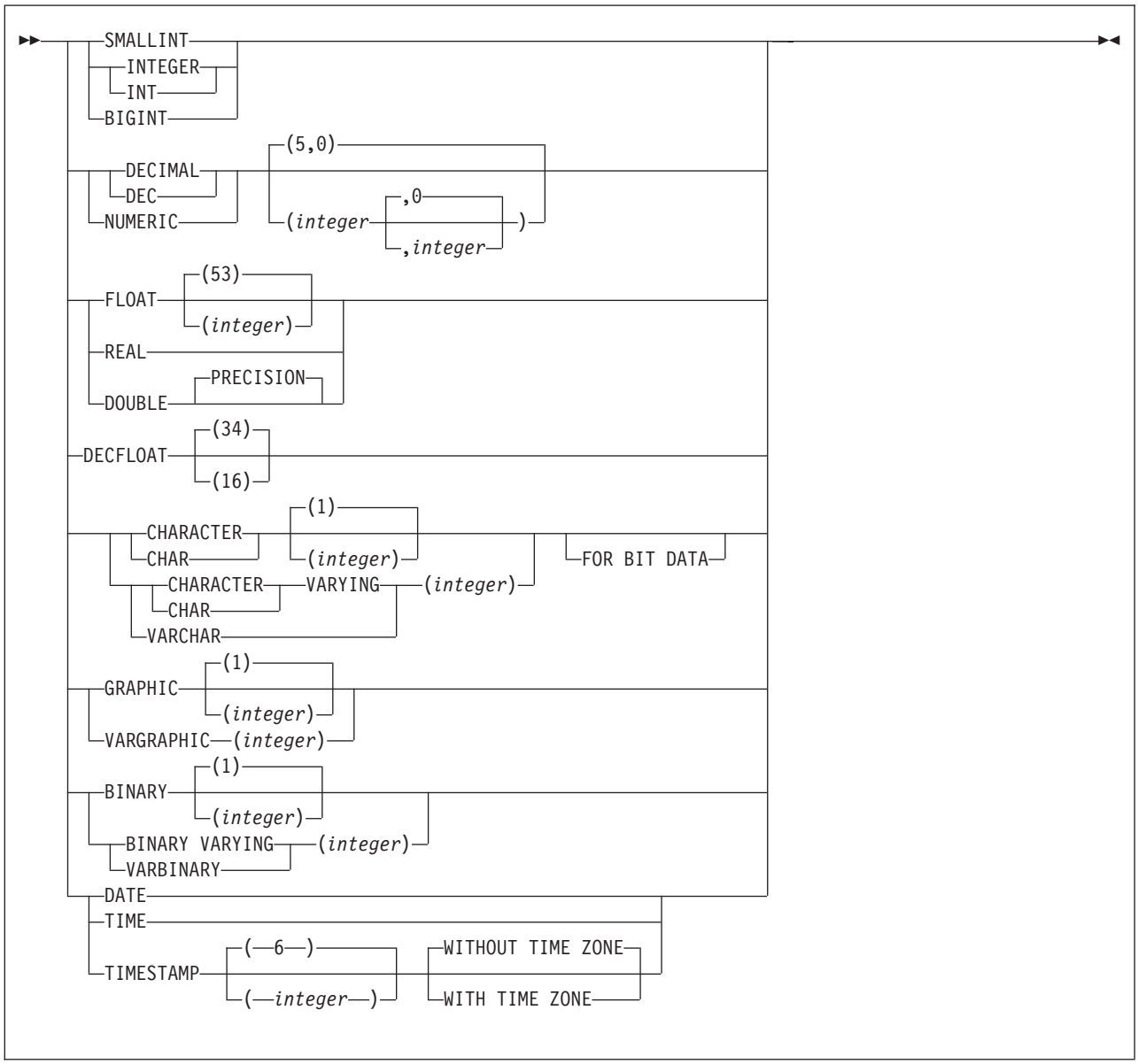
include-column:



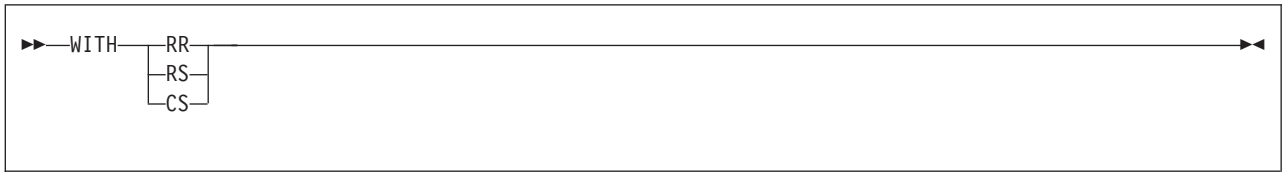
data-type:



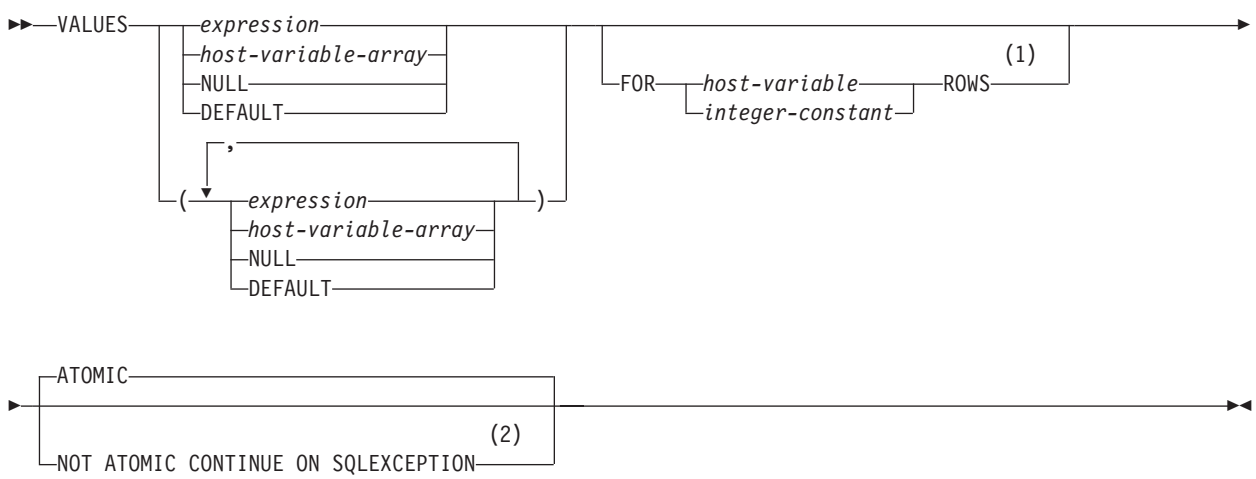
built-in-type:



isolation-clause:



multiple-row-insert:



Notes:

1 The FOR *n* ROWS clause must be specified for a static multiple-row-insert. However, this clause is optional for a dynamic INSERT statement. For a dynamic statement, the FOR *n* ROWS clause is specified on the EXECUTE statement.

2 The ATOMIC or NOT ATOMIC CONTINUE ON SQLEXCEPTION clauses can be specified for a static multiple-row-insert. However, this clause must not be specified for a dynamic INSERT statement. For a dynamic statement, the ATOMIC or NOT ATOMIC CONTINUE ON SQLEXCEPTION clause is specified as an attribute on the PREPARE statement.

Description

INTO *table-name* **or** *view-name*
Identifies the object of the INSERT statement. The name must identify a table or view that exists at the current server. The name must not identify:

- An auxiliary table
- A catalog table
- A read-only view unless an instead of trigger is defined for the insert operation on the view. (For a description of a read-only view, see "CREATE VIEW" on page 1527.)
- A view column that is derived from a constant, expression, or scalar function
- A view column that is derived from the base table column as some other column of the view
- A materialized query table
- A table that is implicitly created for an XML column

In an IMS or CICS application, the DB2 subsystem that contains the identified table or view must be a remote server that supports two-phase commit.

column-name,...
Specifies the columns for which insert values are provided. Each name must identify a column of the table or view. The columns can be identified in any order, but the same column must not be identified more than one time. If extended indicator variables are not enabled, a view column that cannot accept

insert values must not be identified. If extended indicator variables are not enabled, and if the object of the INSERT statement is a view with columns that cannot accept insert values, a list of column names must be specified, and the list must not identify these columns. If a qualifier is specified, it must be valid (that is, the table name must be the table or view name specified after the INTO keyword, and if a qualifier is specified for the table name, it must match the default qualifier).

Omission of the column list is an implicit specification of a list in which every column of the table (that is not defined as implicitly hidden) or view is identified in left-to-right order. This list is established when the statement is prepared and therefore does not include columns that were added to the table after the statement was prepared.

The effect of a rebind on INSERT statements that do not include a column list is that the implicit list of names is re-established. Therefore, the number of columns into which data is inserted can change and cause an error.

include-column
Specifies a set of columns that are included, along with the columns of *table-name* or *view-name*, in the result table of the INSERT statement when it is nested in the FROM clause of the outer fullselect that is used in a subselect, a SELECT statement, or in a SELECT INTO statement. The included columns are appended to the end of the list of columns that is identified by *table-name* or *view-name*.

INCLUDE
Introduces a list of columns that is to be included in the result table of the INSERT statement. The included columns are only available if the INSERT statement is nested in the FROM clause of a SELECT statement or a SELECT INTO statement.

column-name
Specifies the name for a column of the result table of the INSERT statement that is not the same name as another included column nor a column in the table or view that is specified in *table-name* or *view-name*.

data-type
Specifies the data type of the included column. The included columns are nullable.

built-in-type
Specifies a built-in data type. See "CREATE TABLE" on page 1388 for a description of each built-in type.

The CCSID 1208 and CCSID 1200 clauses must not be specified for an include column.

distinct-type
Specifies a distinct type. Any length, precision, or scale attributes for the column are those of the source type of the distinct type as specified by using the CREATE TYPE statement.

OVERRIDING USER VALUE
Specifies that the value specified in the VALUES clause or produced by a fullselect for a column that is defined as either GENERATED ALWAYS or GENERATED BY DEFAULT is ignored. Instead, a system-generated value is inserted, overriding the user-specified value.

If OVERRIDING USER VALUE is specified, the implicit or explicit list of column must include a column that is defined as either GENERATED ALWAYS

or GENERATED BY DEFAULT. For example, a ROWID column, an identity column, or a row change timestamp column.

VALUES
Specifies one new row in the form of a list of values. The number of values in the VALUES clause must be equal to the number of names in the column list and the columns that are identified in the INCLUDE clause. The first value is inserted in the first column in the list, the second value in the second column, and so on. If more than one value is specified, the list of values must be enclosed in parentheses. Assignments to included columns are only processed when the INSERT statement is nested in the FROM clause in a SELECT statement or a SELECT INTO statement.

expression
Any expression of the type described in "Expressions" on page 240. The expression must not include a column name. If *expression* is a host variable, the host variable can identify a structure. Any host variable or structure that is specified must be described in the application program according to the rules for declaring host structures and variables.

If *expression* is a host variable, it can include an indicator variable or an indicator array (in the case of a host structure). When extended indicator variables are enabled, an expression must not be more complex than a reference to a single host variable if the indicator is set to an extended indicator value of default (-5) or unassigned (-7). In addition:

- A CAST specification can be used if either of the following is true:
  - The target column is defined as nullable.
  - The target column is defined as NOT NULL with a non-null default, the source of the CAST specification is a single host variable, and the data attributes (data type, length, precision, and scale) of the host variable are the same as the result of the cast specification.
- A scalar fullselect can be used if either of the following is true for each expression in the select list of the fullselect:
  - The target column that corresponds to the expression is defined as nullable.
  - The expression is not more complex than a reference to a single host variable for which the indicator is set to an extended indicator value of default (-5) or unassigned (-7), or the expression is a CAST specification which would have been valid as a stand-alone expression.

DEFAULT
The default value that is assigned to the column. If the column is a ROWID column, an identity column, a *row-begin* column, a *row-end* column, or a *transaction-start-ID* column, DB2 will generate a value for the column. You can specify DEFAULT only for columns that have an assigned default value, ROWID columns, and identity columns.

For information on default values of data types, see the description of the DEFAULT clause for "CREATE TABLE" on page 1388.

NULL
Specifies the null value as the value of the column. Specify NULL only for nullable columns.

If the implicit or explicit list of columns includes a ROWID, an identity column, or a row change timestamp column that was defined as GENERATED

ALWAYS, you must specify DEFAULT unless you specify the OVERRIDING USER VALUE clause to indicate that any user-specified value will be ignored and a unique system-generated value will be inserted.

For a ROWID or identity column that is defined as GENERATED BY DEFAULT, you can specify a value. However, a value can be inserted into ROWID column defined BY DEFAULT only if a single-column unique index is defined on the ROWID column and the specified value is a valid row ID value that was previously generated by DB2. When a value is inserted into an identity column defined BY DEFAULT, DB2 does not verify that the specified value is a unique value for the column unless the identity column has a single-column unique index.

Although an implicitly hidden DOCID column for XML values is defined as **GENERATED ALWAYS**, you can include the DOCID column in the explicit list of columns and specify a value for it. However, DB2 will ignore the value.

WITH *common-table-expression*
Specifies a common table expression. For an explanation of common table expression, see "common-table-expression" on page 820.

fullselect
Specifies a set of new rows in the form of the result table of a fullselect. If the result table is empty, SQLCODE is set to +100, and SQLSTATE is set to '02000'.

The base object of the INSERT statement and the base object of the fullselect or any subquery of the fullselect can be the same table. In this case, the fullselect is evaluated completely before any rows are inserted.

For an explanation of fullselect, see "fullselect" on page 811.

The number of columns in the result table must be equal to the number of names in the column list and the columns that are identified in the INCLUDE clause. The value of the first column of the result is inserted in the first column in the list, the second value in the second column, and so on. Any values that are produced for a generated column must conform to the rules that are described for those columns under the VALUES clause. Assignments to included columns are only processed when the INSERT statement is nested in the FROM clause of a SELECT statement or a SELECT INTO statement.

If the expression that specifies the value of a result column is a variable, the host variable can include an indicator variable. When extended indicator variables are enabled, the target column that corresponds to an expression in the select list of the fullselect that involves a host variable with an extended indicator value of default (-5) or unassigned (-7), must be defined as nullable and either of the following expressions:

- The expression must not be more complex than a reference to a single host variable.
- The expression must be a CAST specification with the following characteristics:
  - The source of the CAST specification must be a single host variable.
  - The data attributes (data type, length, precision, and scale) of the host variable are the same as the result of the cast specification.

If the object table is self-referencing, the fullselect must not return more than one row.

isolation-clause
Specifies the isolation level that is used when the fullselect is executed.

WITH
Introduces the isolation level, which can be one of the following values:

RR Repeatable read
RS Read stability
CS Cursor stability

The default isolation level of the statement is the isolation level of the package or plan in which the statement is bound, with the package isolation taking precedence over the plan isolation. When a package isolation is not specified, the plan isolation is the default.

QUERYNO *integer*
Specifies the number to be used for this SQL statement in EXPLAIN output and trace records. The number is used for the QUERYNO column of the plan table for the rows that contain information about this SQL statement. This number is also used in the QUERYNO column of the SYSIBM.SYSSTMT and SYSIBM.SYSPACKSTMT catalog tables.

If the clause is omitted, the number that is associated with the SQL statement is the statement number that is assigned during precompilation. Thus, if the application program is changed and then precompiled, that statement number might change.

Using the QUERYNO clause to assign unique numbers to the SQL statements in a program is helpful:
- For simplifying the use of optimization hints for access path selection
- For correlating SQL statement text with EXPLAIN output in the plan table

For information about using optimization hints, such as enabling the system for optimization hints and setting valid hint values, and for information about accessing the plan table, see *DB2 Performance Monitoring and Tuning Guide*.

multiple-row-insert

VALUES
Specifies the items for the rows to be inserted. The number of items in the VALUES clause must equal the number of names in the implicit or explicit column list. The first item in the list provides the value (or values) for the first column in the list. The second item in the list provides the value (or values) for the second column, and so on.

expression
Any expression of the type described in "Expressions" on page 240. The expression must not include a column name. For each row that is inserted, the corresponding column is assigned the value of the expression.

host-variable-array
Each host-variable array must be defined in the application program in accordance with the rules for declaring an array. A host-variable array contains the data for a column of table that is a target of the INSERT. The number of rows to be inserted must be less than or equal to the dimension of each of the host-variable arrays.

An optional indicator array can be specified for each host-variable array. It should be specified if the SQLTYPE of any SQLVAR occurrence indicates

that the SQLVAR is nullable. The indicators must be small integers. The indicator array must be large enough to contain an indicator for each row of input data.

If extended indicator variables are enabled, the extended indicator variable values of DEFAULT or UNASSIGNED can be used inside the indicator array.

DEFAULT
Specifies that the default value is assigned to the column. For each row inserted, the corresponding column is assigned its default value. DEFAULT can be specified only for columns that have a default value. For information on default values of data types, see the description of the DEFAULT clause for "CREATE TABLE" on page 1388.

NULL
Specifies the null value as the value of the column in each row inserted. For each row inserted, the corresponding column is assigned the NULL value. Specify NULL only for nullable columns.

FOR *host-variable* **or** *integer-constant* **ROWS**
Specifies the number of rows to be inserted. For a dynamic INSERT statement, this clause can be specified on the EXECUTE statement. For more information, see the EXECUTE statement. However, this clause is required when a dynamic SELECT statement contains more than one multiple-row INSERT statement.

host-variable or *integer-constant* is assigned to an integral value *k*. If *host-variable* is specified, it must be an exact numeric type with scale zero, and must not include an indicator variable. Furthermore, *k* must be in the range, 0<k<=32767. *k* rows are inserted into the target table from the specified source data.

If a parameter marker is specified in this clause, a value must be provided with the USING clause of the associated EXECUTE or OPEN statement.

ATOMIC or NOT ATOMIC CONTINUE ON SQLEXCEPTION
Specifies whether all of the rows should be inserted as an atomic operation or not.

ATOMIC
Specifies that if the insert for any row fails, all changes made to the database by any of the inserts, including changes made by successful inserts, are undone. This is the default.

NOT ATOMIC CONTINUE ON SQLEXCEPTION
Specifies that, regardless of the failure of any particular insert of a row, the INSERT statement will not undo any changes made to the database by the successful inserts of other rows, and inserting will be attempted for subsequent rows. However, the minimum level of atomicity is at least that of a single insert (that is, it is not possible for a partial insert to complete), including any triggers that might have been executed as a result of the INSERT statement.

This clause is only valid for a static INSERT statement. This clause must also not be specified if the INSERT statement is contained within a SELECT statement. For a dynamic INSERT statement, specify the clause on the PREPARE statement. For more information, see "PREPARE" on page 1781.

Notes

Insert rules:

Insert values must satisfy the following rules. If they do not, or if any other errors occur during the execution of the INSERT statement, no rows are inserted and the position of the cursors are not changed.

- *Default values*. The value inserted in any column that is not in the column list is the default value of the column. Columns without a default value must be included in the column list. Similarly, if you insert into a view, the default value is inserted into any column of the base table that is not included in the view. Hence, all columns of the base table that are not in the view must have a default value.
- *Length*. If the insert value of a column is a number, the column must be a numeric column with the capacity to represent the integral part of the number. If the insert value of a column is a string, the column must be either a string column with a length attribute at least as great as the length of the string, or a datetime column if the string represents a date, time, or timestamp.
- *Assignment*. Insert values are assigned to columns in accordance with the assignment rules described in Chapter 2, "Language elements," on page 53.
- *Uniqueness constraints*. If the identified table or the base table of the identified view has one or more unique indexes, each row inserted into the table must conform to the constraints imposed by those indexes.
- *Referential constraints*. Each nonnull insert value of a foreign key must be equal to some value of the parent key of the parent table in the relationship.
- *Check constraints*. The identified table or the base table of the identified view might have one or more check constraints. Each row inserted must conform to the conditions imposed by those constraints. Thus, each check condition must be true or unknown.
- *Field and validation procedures*. If the identified table or the base table of the identified view has a field or validation procedure, each row inserted must conform to the constraints imposed by that procedure.
- *Indexes with VARBINARY columns*. If the identified table has an index on a VARBINARY column or a column that is a distinct type that is based on VARBINARY data type, that index column cannot specify the DESC attribute. To use the SQL data change operation on the identified table, either drop the index or alter the data type of the column to BINARY and then rebuild the index.
- *Views and the WITH CHECK OPTION*. For views defined with WITH CHECK OPTION, each row you insert into the view must conform to the definition of the view. If the view you name is dependent on other views whose definitions include WITH CHECK OPTION, the inserted rows must also conform to the definitions of those views. For an explanation of the rules governing this situation, see "CREATE VIEW" on page 1527.

  For views that are not defined with WITH CHECK OPTION, you can insert rows that do not conform to the definition of the view. Those rows cannot appear in the view but are inserted into the base table of the view.
- *Omitting the column list*. When you omit the column list, you must specify a value for every column that was present in the table when the INSERT statement was bound or (for dynamic execution) prepared.

- *Triggers*. An INSERT statement might cause triggers to be activated. A trigger might cause other statements to be executed or raise error conditions based on the insert values. If an INSERT statement for a view activates an INSTEAD OF trigger, the validity, referential integrity, and check constraints are checked against the data changes that are performed in the trigger, and not against the definition of the view that activates the trigger or the definition of the underlying tables or views.

  When triggers are processed for an INSERT statement that inserts multiple rows depends on the atomicity option that is in effect for the INSERT statement:

  – **ATOMIC**. The inserts are processed as a single statement. Any statement level triggers are activated one time for the statement, and the transition tables will include all of the rows that were inserted.
  – **NOT ATOMIC CONTINUE ON SQLEXCEPTION**. The inserts are processed separately. Any statement level triggers are processed for each row that is inserted, and the transition table includes the individual row that is inserted. When errors are encountered with this option in effect, processing continues, and some of the specified rows will not be inserted. In this case, if an insert trigger is defined on the underlying base table, the statement level triggers will only be activated for rows that were successfully inserted.

    Regardless of the failure of any particular source row, the INSERT statement will not undo any changes that are made to the database by the statement. Insert will be attempted for rows that follow the failed row. However, the minimum level of atomicity is at least that of a single source row (that is, it is not possible for a partial insert operation to complete), including any triggers that might have been activated as a result of the INSERT statement.

Inserting XML documents:

When XML documents are inserted into a table that contains an XML index, the XML values that are inserted into the index are cast to the data type that is specified on the CREATE INDEX statement. If the XML value cannot be cast to the specified data type, the XML value is ignored for the XML index but the document is still inserted into the table. If the data type that is specified for casting is DECFLOAT, values can be rounded when they are inserted into the index. If the index is unique, the rounding that happens during the cast can result in duplicate values.

Number of rows inserted:

Normally, after an INSERT statement completes execution, the value of SQLERRD(3) in the SQLCA is the number of rows inserted. The value in SQLERRD(3) does not include the number of rows that were inserted as the result of a trigger.

For a complete description of the SQLCA, including exceptions to the above statement, see "SQL communication area (SQLCA)" on page 2069.

Nesting user-defined functions or stored procedures:

An INSERT statement can implicitly or explicitly refer to user-defined functions or stored procedures. This is known as *nesting* of SQL statements. A user-defined function or stored procedure that is nested within the INSERT must not access the table into which you are inserting values.

Locking:

Unless appropriate locks already exist, one or more exclusive locks are acquired at the execution of a successful insert operation. Until a commit

or rollback operation releases the locks, only the application process that performed the insert can access the inserted row. If LOBs are not inserted into the row, application processes that are running with uncommitted read can also access the inserted row. The locks can also prevent other application processes from performing operations on the table. However, application processes that are running with uncommitted read can access locked pages and rows.

Locks are not acquired on declared temporary tables.

Inserting rows into a table with multilevel security :
When you insert rows into a table with multilevel security, DB2 determines the value for the security label column of the row according to the following rules:

- If the user (the primary authorization ID) has write-down privilege or write-down control is not enabled, the user can set the security label for the row to any valid security label. The value that is specified must be assignable to a column that is defined as CHAR(8) FOR SBCS DATA NOT NULL. If the user does not specify a value for the security label or specifies DEFAULT, the security label of the row becomes the same as the security label of the user.
- If the user does not have write-down privilege and write-down control is enabled, the security label of the row becomes the same as the security label of the user.

Inserting rows into a table for which row or column access control is enforced:
When an INSERT statement is issued for a table for which row or column access control is enforced, the rules specified in the enabled row permissions or column masks determine whether the row can be inserted. Typically those rules are based on the authorization ID or role of the process. The following rules describe how the enabled row permissions and column masks are used during INSERT:

- A row to be inserted must not be effected by enabled column masks whose columns are referenced while deriving the source values for the row.

  When a column is referenced while deriving the values of a new row, if the column has an enabled column mask, the masked value is used to derive the new values. If the object table is also column access control activated, the column mask that is applied to derive the new values must ensure that the evaluation of the access control rules defined in the column mask resolves the column to itself, not to a constant or to an expression. If the column mask does not mask the column to itself, the new value cannot be used for insert and an error is returned at run time.

  If the OVERRIDING USER VALUE clause is specified, the corresponding values in the new row are ignored, and the above rule for column masks is not applicable to those values.
- If the row can be inserted, and there is a BEFORE INSERT trigger for the table, the trigger is activated.

  Within the trigger actions, the new values for insert can be modified in the transition variables. When the values return from the trigger, the final values for the new values are the ones for insert.
- A row to be inserted must conform to the enabled row permissions.

  When multiple enabled row permissions are defined for a table, a row access control search condition is derived by application of the logical OR operator to the search condition in each enabled row permission. A

row that conforms to the enabled row permissions is a row that if the row is inserted it can be retrieved back using the row access control search condition.

Column masks are not applicable in this process.

- If the rows can be inserted, and there is an AFTER INSERT trigger for the table, the trigger is activated.

The preceding rules are not applicable to the *include-columns*. The *include-columns* are subject to the rules for the select list because they are not the columns of the object table of the INSERT statement.

Extended indicator variable usage:
When extended indicator variables are enabled, negative indicator values that are outside the range of -1 through -7 must not be specified, and the default and unassigned extended indicator values must not be specified in contexts in which they are not supported.

Extended indicator variables:
In an INSERT statement, a value of unassigned has the effect of setting the column to its default value.

If a target column is not updatable, it must be assigned the extended indicator value of unassigned, unless it is an identity column that is defined as GENERATED ALWAYS. If the target column is an identity column that is defined as GENERATED ALWAYS, it must be assigned the DEFAULT keyword or the extended indicator values of default or unassigned unless the **OVERRIDING USER VALUE** clause is specified.

Extended indicator variables and insert triggers:
The activation of insert triggers is not affected by the use of extended indicator variables. If all columns in the implicit or explicit column list have been assigned an extended indicator value of unassigned or default, an insert where all columns have their respective default values is attempted, and, if successful, the insert trigger is activated.

Extended indicator variables and deferred error checks:
When extended indicator variables are enabled, validation that would otherwise be done in statement preparation (to recognize an insert into a non-updatable column) is deferred until statement execution. If statement validation fails, an error is returned when the statement is run, not at statement preparation.

Table space data compression during an insert operation:
If the table space is defined with the COMPRESS YES option, and data is inserted into a table in the table space, the first rows are stored uncompressed. When a DB2-determined amount of data has been inserted into the table, a compression dictionary is created and stored in the table space. The rows that are inserted into the table after the dictionary is created are stored compressed using the compression dictionary.

Generated columns:
A generated column that is defined as GENERATED ALWAYS should not be specified in the *column-list* unless the corresponding entry in the VALUES list is DEFAULT or an extended indicator that specifies that a default value is to be assigned. Specify the **OVERRIDING USER VALUE** clause to indicate that any user-specified value should be ignored and DB2 should assign the default value when a row is inserted.

Inserting rows into system-period temporal tables:

When a row for a system-period temporal table is inserted, DB2 assigns values to the following columns as indicated:

- A *row-begin* column is assigned a value that is generated by reading the time-of-day clock during execution of the first data change statement in the transaction that requires a value to be assigned to a *row-begin* column or a *transaction-start-ID* column in a table. This also occurs when a row in a system-period temporal table is deleted. DB2 ensures the uniqueness of the generated values for a *row-begin* column across transactions. If multiple rows are inserted within a single SQL transaction, the values for the *row-begin* column are the same for all of the rows and are unique from the values that are generated for the column by another transaction.
- A *row-end* column is assigned the a value for the data type of the column.
- A *transaction-start-ID* column is assigned a unique value per transaction or the null value. The null value is assigned to the *transaction-start-ID* column if the column is nullable. Otherwise, the value is generated by reading the time-of-day clock during execution of the first data change statement in the transaction that requires a value to be assigned to a *row-begin* column or *transaction-start-ID* column in a table. This also occurs when a row in a system-period temporal table is deleted. If multiple rows are inserted within a single SQL transaction, the values for the *transaction-start-ID* column are the same for all the rows and are unique from the values that are generated for the column by another transaction.

If the CURRENT TEMPORAL SYSTEM_TIME special register is set to a non-null value, the underlying target of the INSERT statement cannot be a system-period temporal table. This restriction applies regardless of whether the system-period temporal table is directly or indirectly referenced.

Inserting rows into application-period temporal tables:

When a row is inserted into an application-period temporal table, an error is returned if the period that is defined by the begin column and end column of the application period overlap with the period that is defined by the begin column and end column of the application period for another row in the table.

Inserting rows into archive-enabled tables:

You cannot insert rows into an archive-enabled table if the value of the SYSIBMADM.MOVE_TO_ARCHIVE global variable is Y. Otherwise, if this global variable is not set to Y, you can specify an archive-enabled table as the target of the INSERT statement. In this case, the content of the associated archive table is not affected.

A data change statement must not reference an archive-enabled table when a system-period temporal table or application-period temporal table is also referenced.

INSERT without a column list:

An INSERT statement without a column list does not include implicitly hidden columns, so columns that are defined as implicitly hidden must have a defined default value.

Inserting a row into catalog table SYSIBM.SYSSTRINGS:
If the object table is SYSIBM.SYSSTRINGS, only certain values can be specified, as described in Specifying conversion procedures (DB2 Administration Guide).

Datetime representation when using datetime registers:
As explained in Datetime special registers, when two or more datetime registers are implicitly or explicitly specified in a single SQL statement, they represent the same point in time. This is also true when multiple rows are inserted. When ATOMIC is in effect for the INSERT statement, the special registers are evaluated one time for the processing of the statement. If NOT ATOMIC is in effect, the special registers are evaluated as each row of source data is processed.

Non-atomic processing of an INSERT statement:
When NOT ATOMIC is specified the rows of source data are processed separately. Any references to special registers, sequence expressions, and functions in the INSERT statement are evaluated as each row of source data is processed, Statement level triggers are activated as each row of source data is processed.

If one or more errors occur during the execution of an insert of a row, processing continues. The row that was being inserted at the time of the error is not inserted. Execution continues with the next row to be inserted, and any other changes made during the execution of the multiple-row INSERT statement are not backed out. However, the insert of an individual row is an atomic action.

Diagnostics information for a multiple-row INSERT statement:
A single multiple-row INSERT statement might encounter multiple conditions. These conditions can be errors or warnings. Use the GET DIAGNOSTICS statement to obtain information about all of the conditions that are encountered for one of these INSERT statements. See “GET DIAGNOSTICS” on page 1679 for more information.

If a warning occurs during the execution of an insert of a row, processing continues.

When multiple errors or warnings occur with a non-atomic INSERT statement, diagnostic information for each row is available using the GET DIAGNOSTICS statement. The SQLSTATE and SQLCODE reflect a summary of what happened during the INSERT statement:

- **SQLSTATE 01659, SQLCODE +252.** All rows were inserted, but one or more warnings occurred.
- **SQLSTATE 22529, SQLCODE -253.** At least one row was successfully inserted, but one or more errors occurred. Some warnings might also have occurred.
- **SQLSTATE 22530, SQLCODE -254.** No row was inserted. One or more errors occurred while trying to insert multiple rows of data.
- **SQLSTATE 429BI, SQLCODE -20252.** More errors occurred that DB2 is capable of recording. Statement processing is terminated.

When ATOMIC is in effect, if an insert value violates any constraints or if any other error occurs during the execution of an insert of a row, all changes made during the execution of the multiple-row INSERT statement are backed out. The SQLCA reflects the last warning encountered.

After an INSERT statement that inserts multiple rows of data, both atomic and non-atomic, information is returned to the program through the SQLCA. The SQLCA is set as follows:

- SQLCODE contains the SQLCODE.
- SQLSTATE contains the SQLSTATE.
- SQLERRD3 contains the number of rows actually inserted. SQLERRD3 is the number of rows inserted, if this is less than the number of rows requested, then an error occurred.
- SQLWARN flags are set if they were set during any single insert operation.

The SQLCA is used to return information on errors and warnings found during a multiple-row insert. If indicator arrays are provided, the indicator variable values are used to determine if the value from the host-variable array, or NULL, will be used. The SQLSTATE contains the warning from the last data mapping error.

Specifying the number of rows for a dynamic multiple-row INSERT statement:
Be aware of these considerations when specifying the number of rows to be inserted with a dynamic multiple-row INSERT statement that uses host-variable arrays:

- The FOR n ROWS clause can be specified as part of an INSERT statement or as part of an EXECUTE statement, but not both
- In the INSERT statement, you can specify a numeric constant in the FOR n ROWS clause to indicate the number of rows to be inserted or specify a parameter marker to indicate that the number of rows will be specified with the associated EXECUTE or OPEN statement. A multiple-row INSERT statement that is contained within a SELECT statement must include a FOR n ROWS clause.
- In an EXECUTE statement, when a dynamic INSERT statement is not contained within a SELECT statement, the number of rows can be specified with either the FOR n ROWS clause or the USING clause of the EXECUTE statement:
  - If the INSERT statement did not contain a FOR n ROWS clause, a value for the number of rows to be inserted can be specified in the FOR n ROWS clause of the EXECUTE statement with a numeric constant or host variable.
  - If a parameter marker was specified as part of a FOR n ROWS clause in the INSERT statement, a value for the number of rows must be specified with the USING clause of the EXECUTE statement.
- In an OPEN statement, when a dynamic SELECT statement contains one or more INSERT statements that have FOR n ROWS clauses with parameter markers, the values for the number of rows to be inserted (that is, the values for the parameter markers) must be specified with the USING clause of the OPEN statement.

DRDA considerations for a multiple-row INSERT statement:
DB2 for z/OS limits the size of user data and control information to 10M (except for LOBs, which are processed in a different data stream) for a single multiple-row INSERT statement using host-variable arrays.

Multiple-row insert and fetch statements are supported by any requester or server that supports the DRDA Version 3 protocols. If an attempt is made to issue a multiple-row INSERT or FETCH statement on a server that does not support DRDA Version 3 protocols, an error occurs.

When a multiple-row INSERT statement is executed at a DB2 for z/OS requester, the number of rows being inserted at the requester might not be known in some cases. These cases include:

- The FOR *n* ROWS clause contains a constant value for *n* for either a static or dynamic INSERT statement.
- Host variables are specified on the USING clause of an EXECUTE statement for a dynamic INSERT statement.

In either case, if the number of rows that is being inserted is not known, the requester might flow more data than is required to the server. The number of rows that is actually inserted will be correct because the server knows the correct number of rows to insert. However, performance can be adversely affected. Consider the following scenario:

```
...
long serial_num [10];
struct {
short len;
char data [18];
}name [20]
...
EXEC SQL INSERT INTO T1 VALUES (:serial_num, :name) FOR 5 ROWS
```

At the requester, when this statement is executed, the number of rows being inserted, 5, is not known. As a result, the requester will flow 10 values for serial_num and 10 values for name to the server (because the maximum number of rows that can be inserted without error is 10, which is the size of the smallest host-variable array).

Use the following programming techniques to avoid or minimize problems:

- Avoid using constant values for *n* in the FOR *n* ROWS clause of INSERT statements. For static INSERT statements, this technique ensures that the value for *n* will be known at the requester.
- For dynamic INSERT statements, use the USING DESCRIPTOR clause instead of the USING *host-variables* clause on the EXECUTE statement. If a USING DESCRIPTOR clause is used on the EXECUTE statement, the value for 'n' must be indicated in the DESCRIPTOR.
- If neither of the above methods can be used:
  - Declare your host-variable arrays as small as possible, or indicate that the size of your host-variable arrays are the size of 'n' in your descriptor. This avoids sending large numbers of host-variable-array entries that will not be used to the server.
  - Ensure that varying length string arrays are initialized to a length of 0 (zero). This minimizes the amount of data that is sent to the server.
  - Ensure that decimal host-variable arrays are initialized to valid values. This avoids a negative SQLCODE from being returned if the requester encounters invalid decimal data.

Other SQL statements in the same unit of work:

The following statements cannot follow an INSERT statement in the same unit of work:

- An ALTER TABLE statement that changes the data type of a column (ALTER COLUMN SET DATA TYPE)
- An ALTER INDEX statement that changes the padding attribute of an index with varying-length columns (PADDED to NOT PADDED or vice versa)

Examples

Example 1: Insert values into sample table DSN8B10.EMP.

```
INSERT INTO DSN8B10.EMP
  VALUES ('000205','MARY','T','SMITH','D11','2866',
          '1981-08-10','ANALYST',16,'F','1956-05-22',
         16345,500,2300);
```

Example 2: Assume that SMITH.TEMPEMPL is a created temporary table. Populate the table with data from sample table DSN8B10.EMP.

```
INSERT INTO SMITH.TEMPEMPL
  SELECT *
  FROM DSN8B10.EMP;
```

Example 3: Assume that SESSION.TEMPEMPL is a declared temporary table. Populate the table with data from department D11 in sample table DSN8B10.EMP.

```
INSERT INTO SESSION.TEMPEMPL
  SELECT *
  FROM DSN8B10.EMP
  WHERE WORKDEPT='D11';
```

Example 4: Insert a row into sample table DSN8B10.EMP_PHOTO_RESUME. Set the value for column EMPNO to the value in host variable HV_ENUM. Let the value for column EMP_ROWID be generated because it was defined with a row ID data type and with clause GENERATED ALWAYS.

```
INSERT INTO DSN8B10.EMP_PHOTO_RESUME(EMPNO, EMP_ROWID)
  VALUES (:HV_ENUM, DEFAULT);
```

You can only insert user-specified values into ROWID columns that are defined as GENERATED BY DEFAULT and not as GENERATED ALWAYS. Therefore, in the above example, if you were to try to insert a value into EMP_ROWID instead of specifying DEFAULT, the statement would fail unless you also specify OVERRIDING USER VALUE. For columns that are defined as GENERATED ALWAYS, the OVERRIDING USER VALUE clause causes DB2 to ignore any user-specified value and generate a value instead.

For example, assume that you want to copy the rows in DSN8B10.EMP_PHOTO_RESUME to another table that has a similar definition (both tables have a ROWID columns defined as GENERATED ALWAYS). For the following INSERT statement, the OVERRIDING USER VALUE clause causes DB2 to ignore the EMP_ROWID column values from DSN8B10.EMP_PHOTO_RESUME and generate values for the corresponding ROWID column in B.EMP_PHOTO_RESUME.

```
INSERT INTO B.EMP_PHOTO_RESUME
  OVERRIDING USER VALUE
  SELECT * FROM DSN8B10.EMP_PHOTO_RESUME;
```

Example 5: Assume that the T1 table has one column. Insert a variable (:hv) number of rows of data into the T1 table. The values to be inserted are provided in a host-variable array (:hva).

```
EXEC SQL INSERT INTO T1 FOR :hv ROWS VALUES (:hva:hvind) ATOMIC;
```

In this example, :hva represents the host-variable array and :hvind represents the array of indicator variables.

Example 6: Assume that the T2 table has 2 columns, C1 is a SMALL INTEGER column, and C2 is an INTEGER column. Insert 10 rows of data into the T2 table.

The values to be inserted are provided in host-variable arrays :hva1 (an array of INTEGERS) and :hva2 (an array of DECIMAL(15,0) values). The data values for :hva1 and :hva2 are represented in Table 146:

Table 146. Data values for :hva1 and :hva2

Array entry	:hva1	:hva2
1	1	32768
2	-12	90000
3	79	2
4	32768	19
5	8	36
6	5	24
7	400	36
8	73	4000000000
9	-200	2000000000
10	35	88

```
EXEC SQL INSERT INTO T2 (C1, C2)
  FOR 10 ROWS VALUES (:hva1:hvind1, :hva2:hvind2)
  NOT ATOMIC CONTINUE ON SQLEXCEPTION;
```

After execution of the INSERT statement, the following information will be in the SQLCA:

```
SQLCODE = -253
SQLSTATE = 22529
SQLERRD3 = 8
```

Although an attempt was made to insert 10 rows, only 8 rows of data were inserted. Processing continued after the first failed insert because NOT ATOMIC CONTINUE ON SQLEXCEPTION was specified. You can use the GET DIAGNOSTICS statement to find further information, for example:

```
GET DIAGNOSTICS :num_rows = ROW_COUNT, :num_cond = NUMBER;
```

The result of this statement is num_rows = 8 and num_cond = 2 (2 conditions).

```
GET DIAGNOSTICS CONDITION 2 :sqlstate = RETURNED_SQLSTATE,
                            :sqlcode = DB2_RETURNED_SQLCODE,
                            :row_num = DB2_ROW_NUMBER;
```

The result of this statement is sqlstate = 22003, sqlcode = -302, and row_num = 4.

```
GET DIAGNOSTICS CONDITION 1 :sqlstate = RETURNED_SQLSTATE,
                            :sqlcode = DB2_RETURNED_SQLCODE,
                            :row_num = DB2_ROW_NUMBER;
```

The result of this statement is sqlstate = 22003, sqlcode = -302, and row_num = 8.

Example 7: Assume the above table T2 with two columns. C1 is a SMALL INTEGER column, and C2 is an INTEGER column. Insert 8 rows of data into the T2 table. The values to be inserted are provided in host-variable arrays :hva1 (an array of INTEGERS) and :hva2 (an array of DECIMAL(15,0) values.) The data values for :hva1 and :hva2 are represented in Table 146.

```
EXEC SQL INSERT INTO T2 (C1, C2)
  FOR 8 ROWS VALUES (:hva1:hvind1, :hva2:hvind2)
  NOT ATOMIC CONTINUE ON SQLEXCEPTION;
```

After execution of the INSERT statement, the following information will be in the SQLCA:

```
SQLCODE = -253
SQLSTATE = 22529
SQLERRD3 = 6
```

Although an attempt was made to insert 8 rows, only 6 rows of data were inserted. Processing continued after the first failed insert because NOT ATOMIC CONTINUE ON SQLEXCEPTION was specified. You can use the GET DIAGNOSTICS statement to find further information, for example:

```
GET DIAGNOSTICS :num_rows = ROW_COUNT, :num_cond = NUMBER;
```

The result of this statement is num_rows = 68 and num_cond = 2 (2 conditions).

```
GET DIAGNOSTICS CONDITION 2 :sqlstate = RETURNED_SQLSTATE,
                            :sqlcode = DB2_RETURNED_SQLCODE,
                            :row_num = DB2_ROW_NUMBER;
```

The result of this statement is sqlstate = 22003, sqlcode = -302, and row_num = 4.

```
GET DIAGNOSTICS CONDITION 1 :sqlstate = RETURNED_SQLSTATE,
                            :sqlcode = DB2_RETURNED_SQLCODE,
                            :row_num = DB2_ROW_NUMBER;
```

The result of this statement is sqlstate = 22003, sqlcode = -302, and row_num = 8.

Example 8: Assume that table T1 has two columns. Insert a variable number (:hvn) or rows into T1. The values to be inserted are in host-variable arrays :hva and :hvb. In this example, the INSERT statement is contained within the SELECT statement of cursor CS1. The SELECT statement makes use of two other input host variables (:hv1 and :hv2) in the WHERE clause. Either a static or dynamic INSERT statement can be used.

```
-- Static INSERT statement:
DECLARE CS1 CURSOR WITH ROWSET POSITIONING FOR
   SELECT *
      FROM FINAL TABLE
         (INSERT INTO T1 VALUES (:hva, :hvb) FOR :hvn ROWS)
      WHERE C1 > :hv1 AND C2 < :hv2;
OPEN CS1;
-- Dynamic INSERT statement:
PREPARE INSSTMT FROM
   'SELECT *
       FROM FINAL TABLE
          (INSERT INTO T1 VALUES ( ? , ? ) FOR ? ROWS)
       WHERE C1 > ? AND C2 < ?';
DECLARE CS1 CURSOR WITH ROWSET POSITIONING FOR :INSSTMT;
OPEN CS1 USING :hva, :hvb, :hvn, :hv1, :hv2; (or OPEN CS1 USING DESCRIPTOR ...)
```

If the host-variable arrays for the multiple-row INSERT statement were to be specified using a descriptor, that descriptor (SQLDA) would have to describe all input host variables in the statement, and the order of the entries in the SQLDA should be the same as the order of the order of the host variables, host-variable arrays, and values for the FOR n ROWS clauses in the statement. For example, given the statement above, the SQLVAR entries in the descriptor must be assigned in the following order: :hvn, :hva, :hvb, :hv1, hv2. In addition, the SQLVAR entries for host-variable arrays must be tagged in the SQLDA as column arrays (by specifying a special value in part of the SQLNAME field for a host variable), and the SQLVAR entry for the number of rows value must be tagged in the SQLDA (by specifying another special value in part of the SQLNAME field for the host variable).

Example 9: Insert a row into table T1. The row contains the value 'xyz' for column COL1, and the cardinality of array INTA for column COL2.

```
CREATE TYPE INTARRAY AS INTEGER ARRAY [6];
DECLARE INTA AS INTARRAY;
SET INTA = ARRAY [ 1, 2, 3, 4, 5 ];
CREATE TABLE T1 (COL1 CHAR(7), COL2 INT);
INSERT INTO T1 VALUES ('xyz', CARDINALITY(INTA));
```

Example 10: Insert the values from arrays CHARA and INTA into table T1. For a row of T1, a value of the CHARA array is used for column COL1, and the value of the INTA array with the same array index is used for column COL2.

```
CREATE TYPE INTARRAY AS INTEGER ARRAY[10];
CREATE TYPE CHARARRAY AS CHAR(7) ARRAY[10];
DECLARE INTA AS INTARRAY;
DECLARE CHARA AS CHARARRAY;
SET INTA = ARRAY[1, 2, 3, 4, 5];
SET CHARA = ARRAY['a', 'b', 'c', 'd'];
CREATE TABLE T1 (COL1 CHAR(7), COL2 INT);
INSERT INTO T1
 SELECT *
 FROM UNNEST(CHARA, INTA) AS (COL1, COL2);
```

Example 11: Insert three rows of data into table T1. For each inserted row, assign the value of the tenth element in the INTA array variable to the COL1 column.

```
CREATE TYPE INTARRAY AS INTEGER ARRAY[10];
DECLARE INTA AS INTARRAY;
CREATE VARIABLE VAR1 AS INTEGER;
CREATE VARIABLE VAR2 AS INTEGER;
SET INTA = ARRAY[10, 20, 30, 40, 50, 60, 70, 80, 90, 100];
CREATE TABLE T1 (COL1 INT, COL2 CHAR(10));
SET VAR1 = 10;
SET VAR2 = 3;
-- Perform a multiple row insert (specifying a FOR n ROWS clause).
-- The value to be inserted is specified by a reference to an array element.
INSERT INTO T1 (COL1) VALUES(INTA[VAR1]) FOR VAR2 ROWS;
```

The result of the these operations is that a value of 100 is assigned to column COL1 for three rows.

LABEL

The LABEL statement adds or replaces labels in the descriptions of tables, views, aliases, or columns in the catalog at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

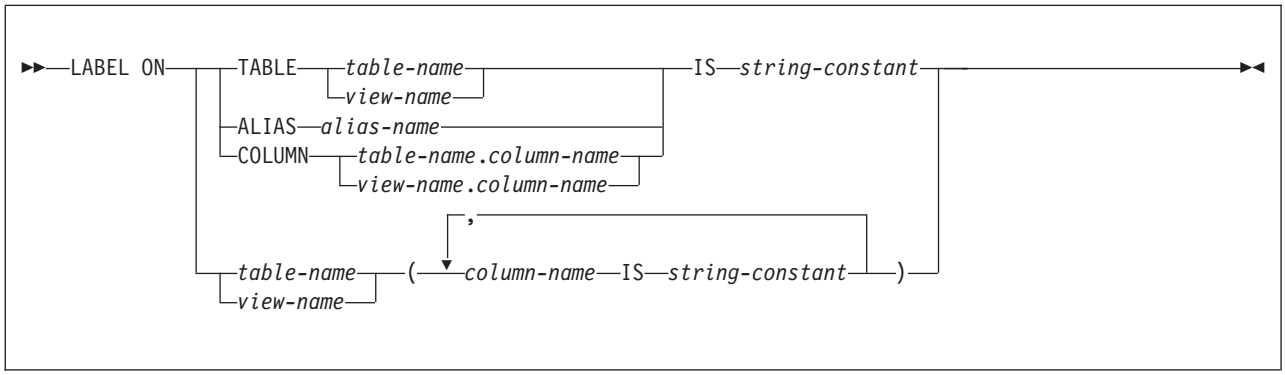
The privilege set that is defined below must include at least one of the following:

- Ownership of the table, view, or alias
- DBADM authority for its database (tables only)
- SYSADM or SYSCTRL authority
- System DBADM

If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set is determined by the DYNAMICRULES behavior in effect (run, bind, define, or invoke) and is summarized in Table 94 on page 841. (For more details on these behaviors, including a list of the DYNAMICRULES bind option values that determine them, see "Authorization IDs and dynamic SQL" on page 75.)

Syntax



Description

TABLE *table-name* **or** *view-name*
Identifies the table or view to which the label applies. The name must identify a table or view that exists at the current server. *table-name* must not identify a declared temporary table. The label is placed into the LABEL column of the SYSIBM.SYSTABLES catalog table for the row that describes the table or view.

ALIAS *alias-name*
Identifies the alias to which the label applies. The name must identify an alias

for a table or view that exists at the current server. The label is placed in the LABEL column of the SYSIBM.SYSTABLES catalog table for the row that describes the alias.

COLUMN *table-name.column-name* **or** *view-name.column-name*
Identifies the column to which the label applies. The name must identify a column of a table or view that exists at the current server. The name must not identify a column of a declared temporary table. The label is placed in the LABEL column of the SYSIBM.SYSCOLUMNS catalog table in the row that describes the column.

Do not use TABLE or COLUMN to define a label for more than one column in a table or view. Give the table or view name and then, in parentheses, a list in the form:

```
column-name IS string-constant,
column-name IS string-constant,...
```

See Example 2 below.

The column names must not be qualified, each name must identify a column of the specified table or view, and that table or view must exist at the current server.

IS Introduces the label you want to provide.

string-constant
Can be any SQL character string constant of up to 30 bytes in length.

Examples

Example 1: Enter a label on the DEPTNO column of table DSN8B10.DEPT.

```
LABEL ON COLUMN DSN8B10.DEPT.DEPTNO
  IS 'DEPARTMENT NUMBER';
```

Example 2: Enter labels on two columns in table DSN8B10.DEPT.

```
LABEL ON DSN8B10.DEPT
 (MGRNO IS 'EMPLOYEE NUMBER FOR THE MANAGER',
  ADMRDEPT IS 'ADMINISTERING DEPARTMENT');
```

LOCK TABLE

The LOCK TABLE statement requests a lock on a table or table space at the current server. The lock is not acquired if the process already holds an appropriate lock.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

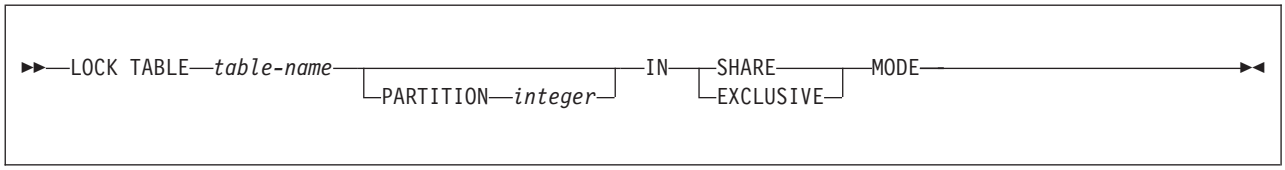
The privilege set that is defined below must include at least one of the following:

- The SELECT privilege on the identified table (the SELECT privilege does not apply to the auxiliary table)
- Ownership of the table
- DBADM authority for the database
- SYSADM or SYSCTRL authority
- DATAACCESS authority

If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set is determined by the DYNAMICRULES behavior in effect (run, bind, define, or invoke) and is summarized in Table 94 on page 841. (For more details on these behaviors, including a list of the DYNAMICRULES bind option values that determine them, see "Authorization IDs and dynamic SQL" on page 75.)

Syntax



Description

table-name
Identifies the table to be locked. The name must identify a table that exists at the current server. It must not identify a view, a temporary table (created or declared), or a catalog table. The lock might or might not apply exclusively to the table. The effect of locking an auxiliary table is to lock the LOB table space that contains the auxiliary table.

PARTITION *integer*
Identifies the partition of a partitioned table space to lock. The table identified by *table-name* must belong to a partitioned table space. The value specified for *integer* must be an integer that is no greater than the number of partitions in the table space.

IN SHARE MODE
For a lock on a table that is not an auxiliary table, requests the acquisition of a lock that prevents other processes from executing anything but read-only operations on the table. For a lock on a LOB table space, IN SHARE mode requests a lock that prevents storage from being reallocated. When a LOB table space is locked, other processes can delete LOBs or update them to a null value, but they cannot insert LOBs with a nonnull value. The type of lock that the process holds after execution of the statement depends on what lock, if any, the process already holds.

IN EXCLUSIVE MODE
Requests the acquisition of an exclusive lock for the application process. Until the lock is released, it prevents concurrent processes from executing any operations on the table. However, unless the lock is on a LOB table space, concurrent processes that are running at an isolation level of uncommitted read (UR) can execute read-only operations on the table.

Notes

Releasing locks: If LOCK TABLE is a static SQL statement, the RELEASE option of bind determines when DB2 releases a lock. For RELEASE(COMMIT), DB2 releases the lock at the next commit point. For RELEASE(DEALLOCATE), DB2 releases the lock when the plan is deallocated (the application ends).

If LOCK TABLE is a dynamic SQL statement, DB2 uses RELEASE(COMMIT) and releases the lock at the next commit point, unless the table or table space is referenced by cached dynamic statements. Caching allows DB2 to keep prepared statements in memory past commit points. In this case, DB2 holds the lock until deallocation or until the commit after the prepared statements are freed from memory. Under some conditions, if a lock is held past a commit point, DB2 demotes the lock state of a segmented table or a nonsegmented table space to an intent lock at the commit point.

Syntax alternatives and synonyms: For compatibility with previous releases of DB2, PART can be specified as a synonym for PARTITION.

Example

Obtain a lock on the sample table named DSN8B10.EMP, which resides in a partitioned table space. The lock obtained applies to every partition and prevents other application programs from either reading or updating the table.

```
LOCK TABLE DSN8B10.EMP IN EXCLUSIVE MODE;
```

Related concepts:

Lock size (DB2 Performance)

The duration of a lock (DB2 Performance)

Related tasks:

Controlling concurrent access to tables (DB2 Performance)

Programming for concurrency (DB2 Performance)

Related reference:

ISOLATION bind option (DB2 Commands)

RELEASE bind option (DB2 Commands)

DYNAMICRULES bind option (DB2 Commands)

MERGE

The MERGE statement updates a target (a table or view, or the underlying tables or views of a fullselect) using the specified input data. Rows in the target that match the input data are updated as specified, and rows that do not exist in the target are inserted. Updating or inserting a row into a view updates or inserts the row into the tables on which the view is based, if no INSTEAD OF trigger is defined on this view.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges that are held by the privilege set that is defined below must include at least one of the following privileges:

- SYSADM authority
- Ownership of the table
- DATAACCESS authority
- If the search condition contains a reference to a column of the table or view, the SELECT privilege for the referenced table or view
- If the insert operation is specified, the INSERT privilege for the table or view
- If the update operation is specified, at least one of the following privileges is required:
  - the UPDATE privilege for the table or view
  - the UPDATE privilege on each column that is updated
  - If the right side of the assignment clause contains a reference to a column of the table or view, the SELECT privilege for the referenced table or view

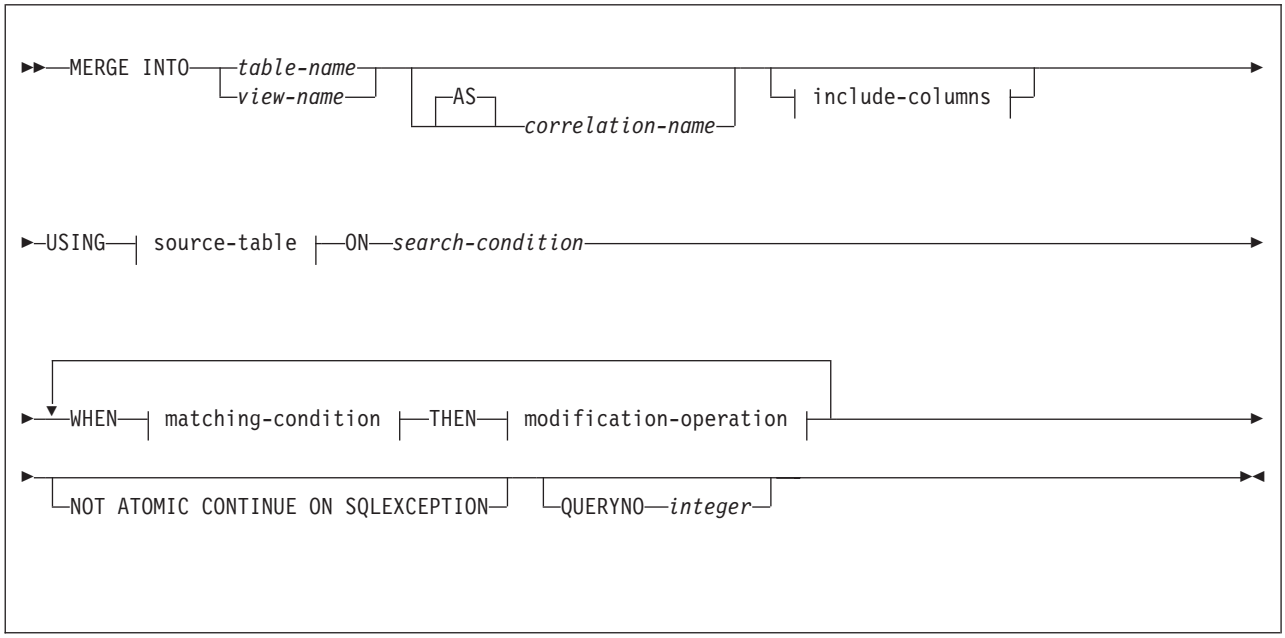
If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

If the insert operation or assignment clause includes a subquery, the privileges that are held by the privilege set must also include at least one of the following privileges:

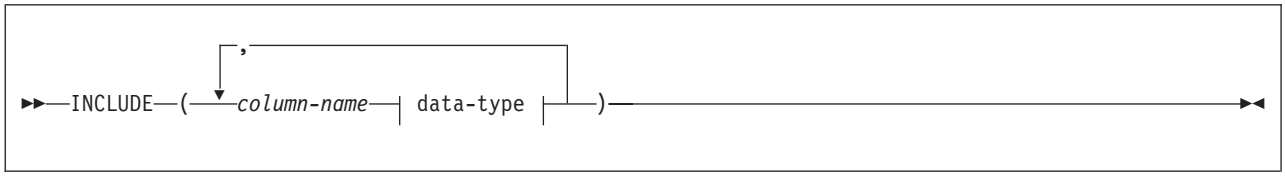
- SYSADM authority
- The SELECT privilege on every table or view that is identified in the subquery
- Ownership of the tables or views that are identified in the subquery
- DATAACCESS authority

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set is determined by the DYNAMICRULES behavior in effect (run, bind, define, or invoke) and is summarized in Table 94 on page 841. (For more information on these behaviors, including a list of the DYNAMICRULES bind option values that determine them, see “Authorization IDs and dynamic SQL” on page 75.)

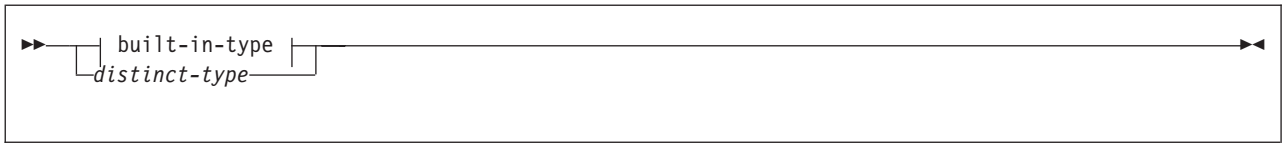
Syntax



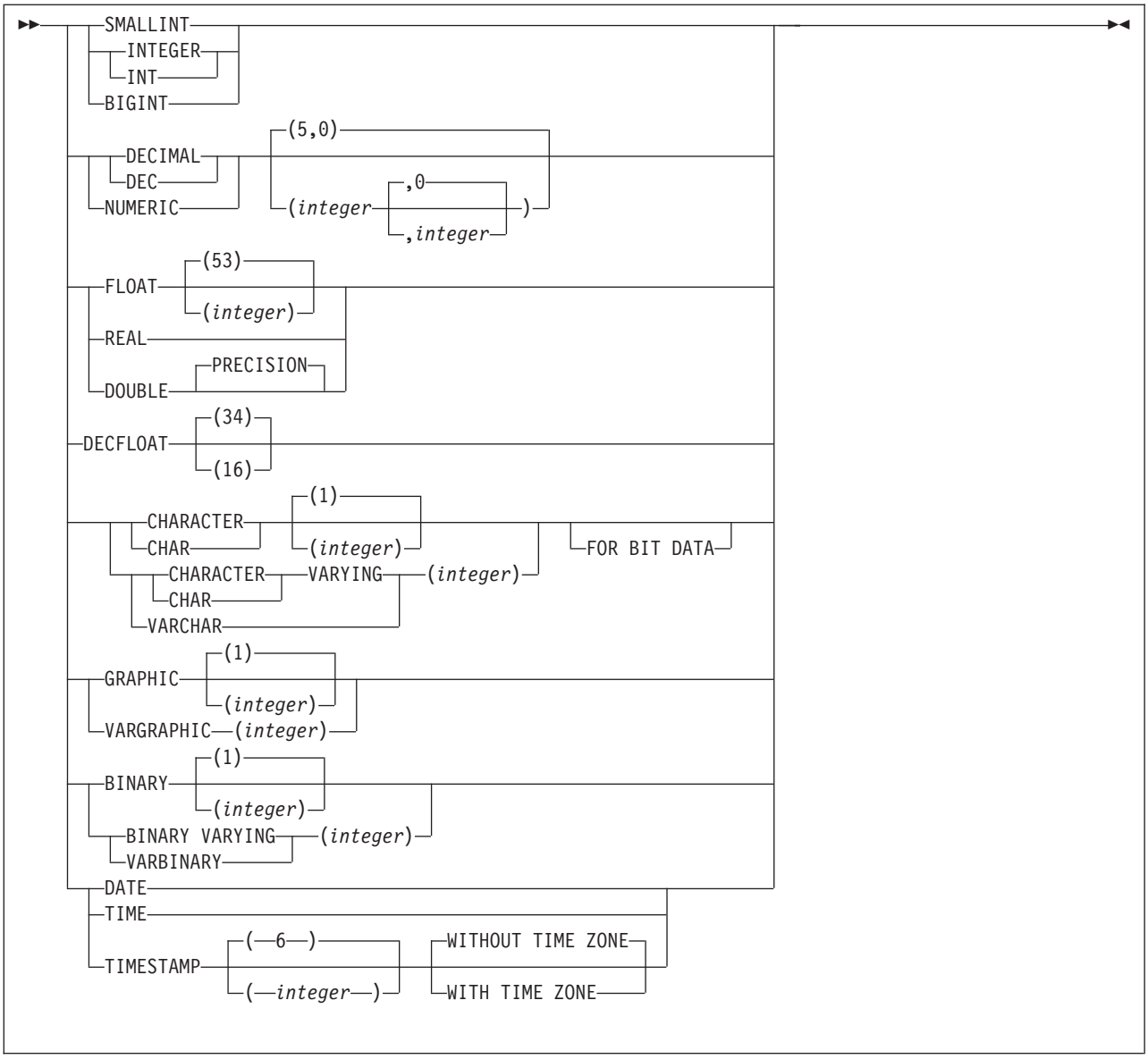
include-columns:



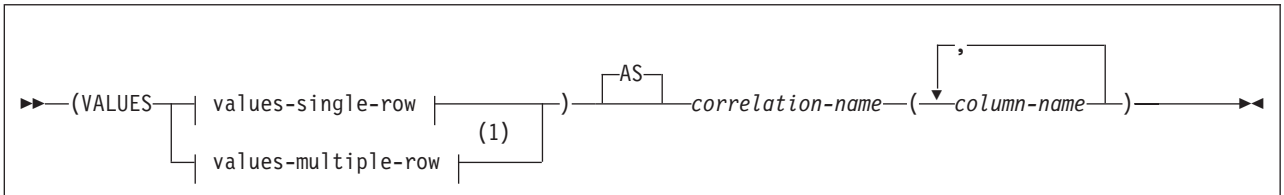
data-type:



built-in-type:



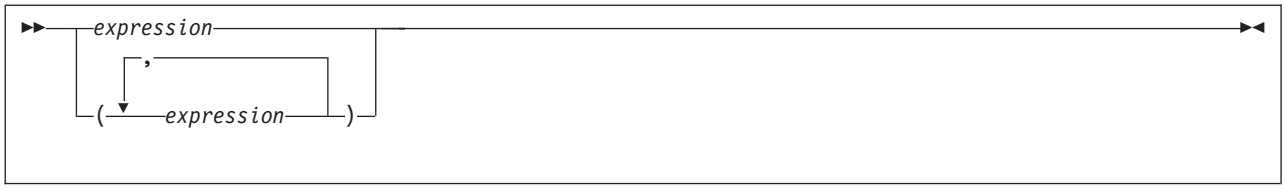
source-table:



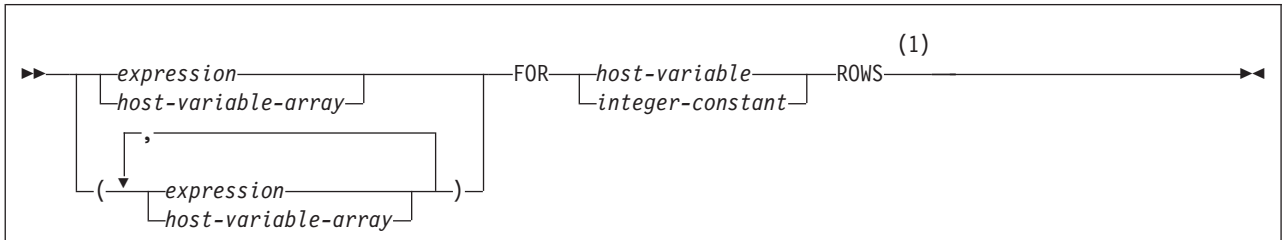
Notes:

1 The NOT ATOMIC clause must be specified when *values-multiple-row* is specified.

values-single-row:



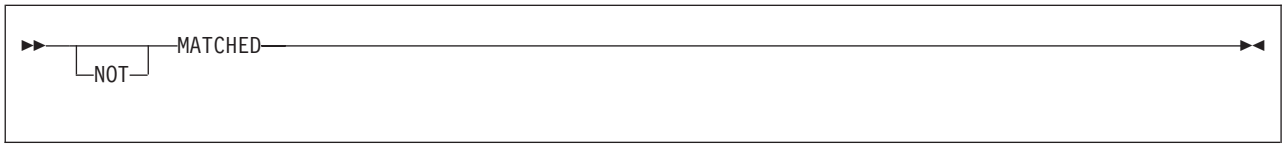
values-multiple-row:



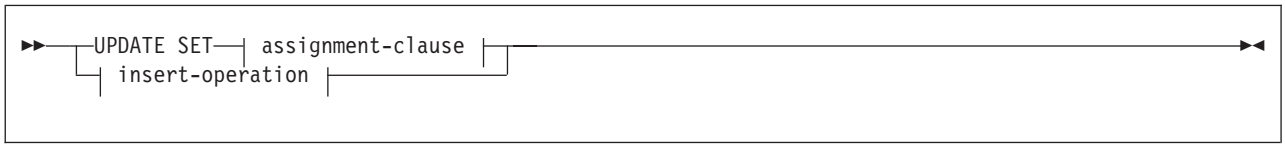
Notes:

1 For a static MERGE statement, if FOR n ROWS is not specified, *values-multiple-row* is treated as *values-single-row*. For a dynamic MERGE statement, FOR n ROWS does not need to be specified in the MERGE statement. It can be specified in the EXECUTE statement, but cannot be specified in both the MERGE and EXECUTE statements.

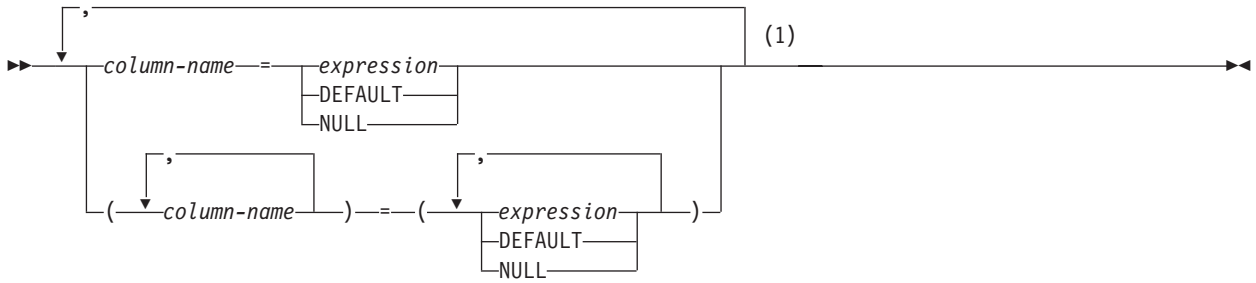
matching-condition:



modification-operation:



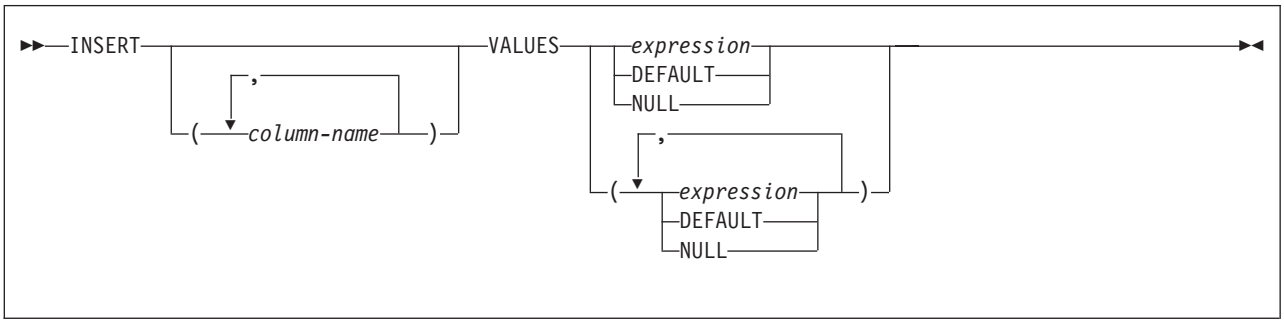
assignment-clause:



Notes:

1 The number of expressions, DEFAULT, and NULL keywords must match the number of column-names.

insert-operation:



Description

INTO *table-name* **or** *view-name*
Identifies the target of the insert or update operations of the MERGE statement. The name must identify a table or view that exists at the current server. The name must not identify:

- A catalog table
- A created global temporary table
- A read-only view
- A system-maintained materialized query table
- A table that is implicitly created for an XML column

If a view is specified as the target of the MERGE statement, the view must not be defined with any INSTEAD OF triggers.

AS *correlation-name*
correlation-name provides an alternate name that can be used when referencing columns of the intermediate result table. If no *correlation-name* is specified, the name of the intermediate result table is the name of the target table or view of the MERGE statement. Otherwise, the name is the *correlation-name*.

include-column
Specifies a set of columns that are included, along with the columns of the specified table or view, in the result table of the MERGE statement when it is nested in the FROM clause of the outer fullselect that is used in a SELECT statement, or in a SELECT INTO statement. The included columns are

appended to the end of the list of columns that are identified by *table-name* or *view-name*. If a value is not specified for an included column, a null value is returned for that column.

INCLUDE
Introduces a list of columns that is to be included in the result table of the MERGE statement. The included columns are only available if the MERGE statement is nested in the FROM clause of a SELECT statement or a SELECT INTO statement. INCLUDE can only be specified when the MERGE statement is nested in the FROM clause of a SELECT statement.

column-name
Specifies the name for a column of the result table of the MERGE statement that is not the same name as another included column or a column in the table or view that is specified in *table-name* or *view-name*.

data-type
Specifies the data type of the included column. The included columns are nullable.

Columns with the following data types can not be used as INCLUDE columns:

- LONG VARCHAR,
- LONG VARGRAPHIC,
- XML
- LOBs
- distinct types that are based on any of the listed data types.

built-in-type
Specifies a built-in data type. See "CREATE TABLE" on page 1388 for a description of each built-in type.

The CCSID 1208 and CCSID 1200 clauses must not be specified for an include column.

distinct-type
Specifies a distinct type. Any length, precision, or scale attributes for the column are those of the source type of the distinct type as specified by using the CREATE TYPE statement.

USING VALUES *values-single-row* **or** *values-multiple-row*
Specifies the values for the row data to merge into the target table or view. *values-single-row* specifies a single row of source data. *values-multiple-row* specifies multiple rows of source data.

expression
Specifies an expression of the type that is described in "Expressions" on page 240. The expression must not include a column name. The expression must not reference a NEXT VALUE or PREVIOUS VALUE expression. If the expression is a single host variable, the host variable can identify a structure. Any host variable or structure that is specified must be described in the application program according to the rules for declaring host structures and variables.

If the expression is a host variable, or if a host variable is being explicitly cast, the host variable can include an indicator variable or an indicator array (in the case of a host structure). Either indicator variables or indicator arrays can be enabled for extended indicator variables.

To provide a null value, specify the NULL keyword on a CAST specification.

host-variable-array
Specifies a host variable array. Each host variable array must be defined in the application program in accordance with the rules for declaring an array. A host variable array contains the data to merge into a target column. The number of rows must be less than or equal to the dimension of each of the host variable arrays. An optional indicator array can be specified for each host variable array. An indicator array should be specified if the SQLTYPE of any SQLVAR occurrence indicates that a column is nullable. The indicator array can be enabled for extended indicator variables. The dimension of the indicator array must be large enough to contain an indicator for each row of input data.

A host structure is not supported in *host-variable-array*.

host-variable-array is supported in C/C++, COBOL, and PL/I.

FOR *host-variable* **or** *integer-constant* **ROWS**
Specifies the number of rows to merge. For a dynamic MERGE statement, this clause can be specified on the EXECUTE statement. *host-variable* or *integer-constant* is assigned to a value *k*. If *host-variable* is specified, it must be an exact numeric type with a scale of zero and must not include an indicator variable. *k* must be in the range of 1 to 32767. *k* rows are merged into the target from the specified source data.

If a parameter marker is specified in FOR *n* ROWS, a value must be provided with the USING clause of the associated EXECUTE statement.

AS *correlation-name*
Specifies a correlation name for the *source-table*.

column-name
Specifies a column name to associate the input data to the SET *assignment-clause* for an update operation or the VALUES clause for an insert operation.

ON *search-condition*
Specifies join conditions between the *source-table* and the target table or view.

Each *column-name* in the search condition must name a column of the target table, view, or *source-table*. A subquery is not allowed in the *search-condition*. If a *column-name* exists in both the target and the *source-table*, the column name must be qualified.

For each row of the *source-table*, the *search-condition* is applied to each row of the target. If the *search-condition* is evaluated as true and the target is not empty, the specified WHEN MATCHED clause is used. Otherwise, the specified WHEN NOT MATCHED clause is used.

WHEN MATCHED or WHEN NOT MATCHED
Specifies the condition under which the *modification-operation* is run.

WHEN MATCHED
Specifies the operation to perform on the rows where the ON *search-condition* is true and the target is not empty. Only UPDATE can be specified after the THEN clause. WHEN MATCHED must not be specified more than one time.

WHEN NOT MATCHED
Specifies the operation to perform on the rows where the ON

search-condition is false or unknown, or the target is empty. Only INSERT can be specified after the THEN clause. WHEN NOT MATCHED must not be specified more than one time.

THEN *update-operation* **or THEN** *insert-operation*
Specifies the operation to run when the *matching-condition* evaluates to true.

UPDATE SET
Specifies the update operation to run when the *matching-condition* evaluates to true.

When extended indicator variables are enabled, a column of the source table must not be referenced multiple times in a single modification-operation. Extended indicator variables are enabled when EXTENDEDINDICATOR(YES) is used, or when the WITH EXTENDED INDICATORS prepare attribute has been specified for the MERGE statement.

The rows that are updated from a *source-row* are subject to more updates by subsequent *source-rows* in the same statement. The update is cumulative.

An update-operation in a MERGE statement will not rest the AREO* status on a table.

assignment-clause
Specifies a list of column updates.

column-name
Identifies a column to update. *column-name* must identify a column of the specified table or view, and that column must be updatable. The column must not be a generated column, or a column of a view that is derived from a scalar function, a constant, or a expression. *column-name* can also identify an included column. The same *column-name* must not be specified more than one time.

Assignments to included columns are only processed when the MERGE statement is nested in the FROM clause of a SELECT statement or a SELECT INTO statement. There must be at least one assignment clause that specifies a *column-name* that is not an included column. A view column that is derived from the same column as another column of the view can be updated, but both columns cannot be updated in the same MERGE statement.

expression
Specifies the new value of the column. The expression is any expression of the type that is described in "Expressions" on page 240. The expression must not include an aggregate function.

An expression can contain references to columns of *source-table* or target. A column name is first checked as a column of the target, and then checked as a column of the source table. For each row that is updated, the value of a target column in an expression is the value of the column in the row before the row is updated. *expression* cannot contain references to an included column.

If *expression* is a reference to a single column of the *source-table*, the *source-table* column value might have been specified with an extended indicator variable value. The effects of such indicator variables apply to the corresponding target columns of the *assignment-clause*.

When extended indicator variables are enabled, an expression must not be more complex than a reference to a single column of the source

table, or a single host variable if the indicator is set to an extended indicator value of default (-5) or unassigned (-7). In addition, a CAST specification can be used if either:

- The target column is defined as nullable.
- the target column is defined as NOT NULL with a non-null default, the source of the CAST specification is a single host variable, and the data attributes (data type, length, precision, and scale) of the host variable are the same as the result of the cast specification.

DEFAULT
Specifies the default value for the column. The value that is assigned depends on how the column is defined.

A ROWID column must not be set to the DEFAULT keyword.

An identity column or a row change timestamp column that is defined as GENERATED ALWAYS can be set only to the DEFAULT keyword.

If the column is defined using the NOT NULL clause and the GENERATED clause is not used, or the WITH DEFAULT clause is not used, the DEFAULT keyword cannot be specified for that column.

NULL
Specifies the null value as the new value of the column. Specify NULL only for nullable columns.

insert-operation
Specifies the insert operation to run for the rows where the *matching-condition* evaluates to true.

The rows that are inserted from a source-row are immediately subject for update by subsequent source-rows in the same statement.

INSERT
Specifies a list of column names and row value expressions to use of the insert operation.

The number of values for the row in the row-value expression must be equal to the number of names in the insert column list. The first value is inserted into the first column in the list, the second value into the second column, and so on.

column-name

Specifies the columns for which the insert values are provided. Each name must identify a column of the table or view

If an included column is not specified in the list of column names, the value of the included column is set to null. The column list cannot contain only included columns.

The same column must not be specified more than one time. A view column that cannot accept insert values must not be specified. A value cannot be inserted into a view column that is derived from one of the following:

- a constant, an expression, or a scalar function
- the same column of the base table as another column of the view

If the object of the operation is a view that contains columns that cannot accept insert values, a list of column names must be specified and the list must not specify these columns.

Omission of the column list is an implicit specification of a list in which every column of the table (that is not defined as implicitly hidden) or view is identified in left-to-right order. This list is established when the statement is prepared and therefore does not include columns that were added to the table after the statement was prepared.

VALUES
Introduces one or more rows of values to insert.

expression
Specifies an expression of the type that does not include a column name of the target. If *expression* is a host variable, the host variable can include an indicator variable, or in the case of a host structure, an indicator array. When extended indicator variables are enabled, an expression must not be more complex than a reference to a single host variable if the indicator is set to an extended indicator value of default (-5) or unassigned (-7).

In addition, a CAST specification can be used if either:

- The target column is defined as nullable.
- the target column is defined as NOT NULL with a non-null default, the source of the CAST specification is a single host variable, and the data attributes (data type, length, precision, and scale) of the host variable are the same as the result of the cast specification.

DEFAULT
Specifies to assign the default value to the column. DEFAULT must only be specified for columns that have a default value. If the column is specified in the INCLUDE column list, the column value is set to null.

DEFAULT must be specified for a column that is defined as GENERATED ALWAYS. A valid value can be specified for a column that is defined as GENERATED BY DEFAULT.

NULL
Specifies the null value as the value of the column. Specify NULL only for nullable columns.

NOT ATOMIC CONTINUE ON SQLEXCEPTION

The rows of input data are processed separately. Any statement level triggers are processed for each row of source data that is processed, and the transition table includes the individual row that was processed. When errors are encountered and this option is in effect, processing continues, and some of the specified rows will not be processed. In this case, if an appropriate trigger is defined on the underlying base table, the statement level trigger will only be activated for rows that were successfully processed.

Regardless of the failure of any particular source row, the MERGE statement will not undo any changes that are made to the database by the statement. Merge will be attempted for rows that follow the failed row. However, the minimum level of atomicity is at least that of a single source row (that is, it is not possible for a partial merge to complete), including any triggers that might have been activated as a result of the MERGE statement.

QUERYNO *integer*
Specifies the number for this SQL statement that is used in EXPLAIN output

and trace records. The number is used for the QUERYNO column of the plan table for the rows that contain information about this SQL statement. This number is also used in the QUERYNO column of the SYSIBM.SYSSTMT and SYSIBM.SYSPACKSTMT catalog tables.

If QUERYNO is not specified, the number that is associated with the SQL statement is the statement number that is assigned during precompilation. Thus, if the application program is changed and then precompiled, the statement number might change.

Notes

SQLCA and GET DIAGNOSTICS considerations:
The GET DIAGNOSTICS statement can be used immediately after the MERGE statement to check which input rows fail during the merge operation. The GET DIAGNOSTICS statement information item, NUMBER, indicates the number of conditions that are raised. The GET DIAGNOSTICS condition information item, DB2_ROW_NUMBER, indicates the input source rows that cause an error.

Trigger considerations:
A MERGE statement might cause triggers to be activated. A trigger might cause other statements to be executed or raise error conditions depending on the source data values. A before-update or before-insert trigger processes immediately before the update or insert operation.

If a source row results in an insert, any after-insert triggers are activated after the insert operation completes.

If a source row results in updates, any after-update triggers are activated after all of the update operations complete.

Indexes with VARBINARY columns:
Suppose that the identified table has an index on a VARBINARY column or a column that is a distinct type that is based on VARBINARY data type. In that case, that index column cannot specify the DESC attribute. To use the SQL data change operation on the identified table, either drop the index or alter the data type of the column to BINARY and then rebuild the index.

Considerations for a MERGE without a column list in insert-operation:
A MERGE statement without a specified column list as part of *insert-operation* does not include implicitly hidden columns. Therefore, such columns must have a defined default value.

Considerations for non-atomic processing of a MERGE statement:
When NOT ATOMIC is specified, the rows of source data are processed separately. Any references to special registers, sequence expressions, and functions in the MERGE statement are evaluated as each row of source data is processed. Statement level triggers are activated as each row of source data is processed.

If one or more errors occur during the operation for a row of source data, processing continues. The row that was being processed at the time of the error is not inserted or updated. Execution continues with the next row to be processed, and any other changes that are made during the execution of the multiple-row MERGE statement are not backed out. However, the processing of an individual row is an atomic action.

DRDA considerations:

DB2 Connect™ Version 9.1 and subsequent releases support the MERGE statement. The support is for CLI only, with no embedded static SQL support.

When you run a MERGE statement at a DB2 for z/OS requester, cases might exist where the requestor does not know the number of rows in the source table. This situation includes the following cases:

- For static or dynamic MERGE statements, of the FOR *n* ROWS clause contains a constant value for *n*.
- For dynamic MERGE statements, of host variables are specified on the USING clause of an EXECUTE statement.

For both of these cases, if the number of rows in the source table is not known, the requester might send more data than is required to the server. The number of rows that are processed is correct because the server knows the correct numbers of rows to process. However, performance might be adversely affected. Consider the following example:

```
...long serial num [10];
struct { short len;
char data [18];
       }
name[20]...
EXEC SQL
MERGE INTO T1
  USING (VALUES (:serial_num, :name))
  FOR 5 ROWS...
```

When this statement is run at the requester, the number of rows to merge (five) is not known. As a result, the requester sends 10 values for `serial-name and name` to the server because 10 is the size of the smallest host variable array and is, therefore, the maximum number of rows that can merge without causing an error.

Do the following to help minimize performance problems:

- Avoid using numeric constants in the FOR *n* ROWS clause of the MERGE statement. For static MERGE statements, avoiding numeric constants ensures that the values for *n* are known at the requester.
- For dynamic MERGE statements, use the USING DESCRIPTOR clause instead of the USING *host-variable* clause on the EXECUTE statement. If a USING DESCRIPTOR clause is used on the EXECUTE statement, the value for n must be indicated in the descriptor.
- If either of the previous methods cannot be used, perform the following actions:
  - Make your host variable arrays as small as possible, or declare that the size of your host variable arrays are the size of n in the descriptor. This action avoids sending many unused host variable array entries to the server.
  - Ensure that varying length string arrays are initialized to a length of 0 (zero). Doing so minimizes the amount of data that is sent to the server.
  - Ensure that decimal host variable arrays are initialized to valid values. Doing so causes the requester to avoid sending a negative SQLCODE if the requester encounters invalid decimal data.

Extended indicator variable usage:

When extended indicator variables are enabled, negative indicator values

outside the range of -1 through -7 must not be specified. Also, the default and unassigned extended indicator values must not be used in contexts in which they are not supported.

Extended indicator variables in the *assignment-clause*:
Assigning the extended indicator a value of unassigned leaves the target column set to its current value, as if it had not been specified in the statement. Assigning the extended indicator a value of default assigns the default value to the column.

If a target column is not updatable, for example an identity column that is defined as GENERATED ALWAYS, it must be assigned the extended indicator value of unassigned.

The *assignment-clause* must not assign all target columns to an extended indicator value of unassigned.

Extended indicator variables in the *insert-operation*:
In *insert-operation*, a value of unassigned sets the column to its default value.

If a target column is not updatable, it must be assigned the extended indicator value of unassigned. The exception is an identity column that is defined as GENERATED ALWAYS. If the target column is an identity column that is defined as GENERATED ALWAYS, it must be assigned the DEFAULT keyword or the extended indicator value of default or unassigned.

Extended indicator variables and update triggers:
If a target column is assigned an extended indicator value of unassigned, that column is not considered to have been updated. That column is treated as if it had not been specified in the **OF** *column-name* list of any update trigger that is defined on the target table.

Extended indicator variables and insert triggers:
The activation of insert triggers is not affected by the use of extended indicator variables. Suppose that all columns in the implicit or explicit column list are assigned an extended indicator value of unassigned or default. Then, assume that an insert operation where all columns are assigned to the respective default values is attempted. If that operation is successful, the insert trigger is activated.

Extended indicator variables and deferred error checks:
When extended indicator variables are enabled, validation that would otherwise be done in statement preparation (to recognize an insert into or an update of a non-updatable column) is deferred until statement execution. If validation fails, an error is returned at execution instead of statement preparation.

Table space data compression during an insert operation:
If the table space is defined with the COMPRESS YES option, and data is inserted into a table in the table space, the first rows are stored uncompressed. When a amount of data that is determined by DB2 is inserted into the table, a compression dictionary is created and stored in the table space. The rows that are inserted into the table after the dictionary is created are stored compressed by using the compression dictionary.

System-period temporal tables:

When a MERGE statement is processed for a system-period temporal table, the rows are affected in the same way as if the specific data change operation was invoked.

Archive-enabled tables:

Consider the case when the target of a MERGE statement is an archive-enabled table, and the merge operation includes an insert or update operation. In this case, the involved rows are affected in the same way as if the insert or update operation was directly invoked on the table.

Related information:

"INSERT" on page 1734

"UPDATE" on page 1933

Tables with enforced row and column access controls:

For information about how enabled row permissions and column masks affect the update and insert operations in the MERGE statement, see the INSERT and UPDATE statement information.

Related information:

"INSERT" on page 1734

"UPDATE" on page 1933

Examples

Example 1: Update the descriptions for activities that exist in the RECORDS table. Otherwise, insert the activity and its description into the RECORDS table.

```
MERGE INTO RECORDS AR
  USING (VALUES (:hv_activity, :hv_description)
    FOR :hv_nrows ROWS)
    AS AC (ACTIVITY, DESCRIPTION)
  ON (AR.ACTIVITY = AC.ACTIVITY)
  WHEN MATCHED THEN UPDATE SET DESCRIPTION = AC.DESCRIPTION
  WHEN NOT MATCHED THEN INSERT (ACTIVITY, DESCRIPTION)
     VALUES (AC.ACTIVITY, AC.DESCRIPTION)
  NOT ATOMIC CONTINUE ON SQLEXCEPTION;
```

Example 2: Use the transaction data to merge rows into the account table. Update the balance from the transaction data against an account ID and insert new accounts from the transaction data where the accounts do not already exist.

```
MERGE INTO ACCOUNT AS A
  USING (VALUES (:hv_id, :hv_amount)
    FOR 3 ROWS)
    AS T (ID, AMOUNT)
  ON (A.ID = T.ID)
  WHEN MATCHED THEN UPDATE SET BALANCE = A.BALANCE + T.AMOUNT
  WHEN NOT MATCHED THEN INSERT (ID, BALANCE)
      VALUES (T.ID, T.AMOUNT)
  NOT ATOMIC CONTINUE ON SQLEXCEPTION;
```

Example 3: Update the list of activities that are organized by group A in the RECORDS table. Update the activities information (description and date when last modified) in the RECORDS table if the activities exist in the RECORDS table and are also organized by group A. Insert new activities into the RECORDS table.

```
-- hv_nrows = 3
-- hv_activity(1) = 'D'; hv_description(1) = 'Dance'; hv_date(1) = '03/01/07'
-- hv_activity(2) = 'S'; hv_description(2) = 'Singing'; hv_date(2) = '03/17/07'
-- hv_activity(3) = 'T'; hv_description(3) = 'Tai-chi'; hv_date(3) = '05/01/07'
```

```
-- hv_group = 'A';
-- note that hv_group is not an array. All 3 values contain the same values
MERGE INTO RECORDS AR
  USING (VALUES (:hv_activity, :hv_description, :hv_date, :hv_group)
     FOR :hv_nrows ROWS)
     AS AC (ACTIVITY, DESCRIPTION, DATE, GROUP)
  ON AR.ACTIVITY = AC.ACTIVITY AND AR.GROUP = AC.GROUP
  WHEN MATCHED
  THEN UPDATE SET (DESCRIPTION, DATE, LAST_MODIFIED)
                  = (AC.DESCRIPTION, AC.DATE, CURRENT TIMESTAMP)
  WHEN NOT MATCHED
  THEN INSERT (GROUP, ACTIVITY, DESCRIPTION, DATE, LAST_MODIFIED)
     VALUES (AC.GROUP, AC.ACTIVITY, AC.DESCRIPTION, AC.DATE, CURRENT TIMESTAMP)
  NOT ATOMIC CONTINUE ON SQLEXCEPTION;
```

Example 4: Use two arrays, CHARA and INTA, as input to a MERGE statement. Column COL2 is set to the cardinality of CHARA for matching rows, and COL2 is set to the cardinality of INTA for non-matching rows.

```
CREATE TYPE INTARRAY AS INTEGER ARRAY[6];
CREATE TYPE CHARARRAY AS CHAR(20) ARRAY[7];
DECLARE INTA AS INTARRAY;
DECLARE CHARA AS CHARARRAY;
CREATE VARIABLE SI INT;
SET CHARA = ARRAY['a', 'b', 'c'];
SET INTA = ARRAY [1, 2, 3, 4, 5];
CREATE TABLE T1 (COL1 CHAR(7), COL2 INT);
INSERT INTO T1 VALUES ('abc', 10);
MERGE INTO T1 AS A
 USING TABLE (VALUES ('rsk', 3 ) ) AS T (ID, AMOUNT)
 ON A.COL1 = T.ID
 WHEN MATCHED
  THEN UPDATE SET COL2 = CARDINALITY(CHARA)
 WHEN NOT MATCHED
  THEN INSERT (COL1, COL2 ) VALUES (T.ID, CARDINALITY(INTA));
```

OPEN

The OPEN statement opens a cursor so that it can be used to process rows from its result table.

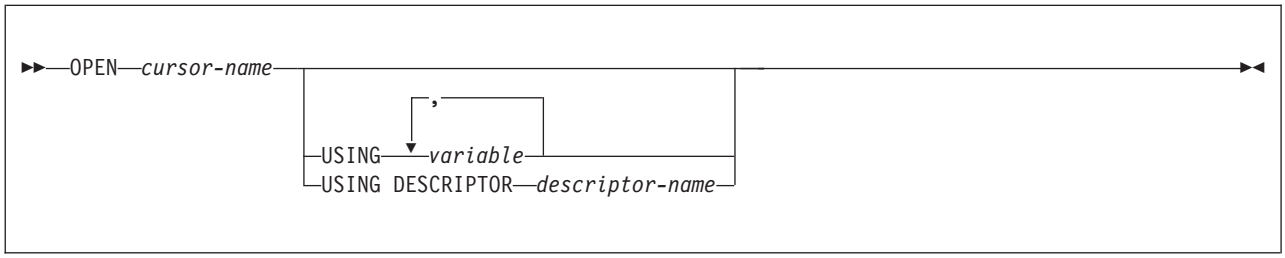
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

See "DECLARE CURSOR" on page 1535 for the authorization required to use a cursor.

Syntax



Description

cursor-name
Identifies the cursor to be opened. The *cursor-name* must identify a declared cursor as explained in "DECLARE CURSOR" on page 1535. When the OPEN statement is executed, the cursor must be in the closed state.

The SELECT statement of the cursor is either one of the following types of SELECT statements:

- The *select-statement* that is specified in the DECLARE CURSOR statement
- The prepared *select-statement* that is identified by the *statement-name* that is specified in the DECLARE CURSOR statement.

If the statement has not been successfully prepared, or is not a *select-statement*, the cursor cannot be successfully opened.

The result table of the cursor is derived by evaluating the SELECT statement. The evaluation uses the current values of any special registers or PREVIOUS VALUE expressions that are specified in the SELECT statement, and the current values of any host variables that are specified in the SELECT statement or the USING clause of the OPEN statement. The rows of the result table can be derived during the execution of the OPEN statement, and a temporary copy of a result table can be created to hold those rows. They can be derived during the execution of later FETCH statements. In either case, the cursor is placed in the open state and positioned before the first row of its result table.

If the table is empty, the position of the cursor is effectively "after the last row." The DB2 system does not indicate an empty table when the OPEN statement is executed. A subsequent fetch for the cursor might return the SQLSTATE warning of '02000'.

USING

Introduces a list of host variables whose values are substituted for the parameter markers (question marks) or host variables in the statement of the cursor, depending on the declaration of the cursor:

- If the DECLARE CURSOR statement included *statement-name*, the statement was prepared with a PREPARE statement. The host variables specified in the USING clause of the OPEN statement replace any parameter markers in the prepared statement. This reflects the typical use of the USING clause of the OPEN statement For an explanation of parameter marker replacement, see "PREPARE" on page 1781.

  If the prepared statement includes parameter markers, you must use USING. If the prepared statement does not include parameter markers, USING is ignored.
- If the DECLARE CURSOR statement included *select-statement* and the SELECT statement included host variables, the USING clause of the OPEN statement can be used to specify host variables that are to override the values that were specified when the cursor was defined. In this case, the OPEN statement is executed as if each host variable in the SELECT statement were a parameter marker except that the attributes of the target variable are the same as the host variables in the SELECT statement. The effect is to override the values of the host variables in the SELECT statement of the cursor with the values of the host variables specified in the USING clause. The overriding value is always the value of the main variable because indicator variables are ignored in this context without warning.

variable

Identifies a variable or a host structure that is declared in the application program in accordance with the rules for declaring variables and host structures. When the statement is executed, a reference to a structure is replaced by a reference to each of its variables. The number of variables must be the same as the number of parameter markers in the prepared statement. The *n*th variable corresponds to the *n*th parameter marker in the prepared statement. Where appropriate, locator variables and file reference variables can be provided as the source of values for parameter markers.

DESCRIPTOR *descriptor-name*

Identifies an SQLDA that contains a valid description of the input host variables.

Before the OPEN statement is processed, the user must set the following fields in the SQLDA:

- SQLN to indicate the number of SQLVAR occurrences provided in the SQLDA

  A REXX SQLDA does not contain this field.
- SQLABC to indicate the number of bytes of storage allocated for the SQLDA
- SQLD to indicate the number of variables used in the SQLDA when processing the statement
- SQLVAR occurrences to indicate the attributes of the variables

The SQLDA must have enough storage to contain all SQLVAR occurrences. If LOBs or distinct types are present in the result table, there must be additional SQLVAR entries for each input host variable. For more information on the SQLDA, which includes a description of the SQLVAR and an explanation on how to determine the number of SQLVAR occurrences, see "SQL descriptor area (SQLDA)" on page 2079.

SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN. It must be the same as the number of parameter markers in the prepared statement.

See "Identifying an SQLDA in C or C++" on page 2099 for how to represent *descriptor-name* in C.

Notes

Errors occurring on OPEN: In local and remote processing, the DEFER(PREPARE) and REOPT(ALWAYS)/REOPT(ONCE) bind options can cause some SQL statements to receive "delayed" errors. For example, an OPEN statement might receive an SQLCODE that normally occurs during PREPARE processing. Or a FETCH statement might receive an SQLCODE that normally occurs at OPEN time.

Closed state of cursors: All cursors in an application process are in the closed state when:

- The application process is started.
- A new unit of work is started for the application process unless the WITH HOLD option has been used in the DECLARE CURSOR statement.
- The application was precompiled with the CONNECT(1) option (which implicitly closes any open cursors).

A cursor can also be in the closed state because:

- A CLOSE statement was executed.
- An error was detected that made the position of the cursor unpredictable.

To retrieve rows from the result table of a cursor, you must execute a FETCH statement when the cursor is open. The only way to change the state of a cursor from closed to open is to execute an OPEN statement.

Effect of a temporary copy of a result table: DB2 can process a cursor in two different ways:

- It can create a temporary copy of the result table during the execution of the OPEN statement. You can specify INSENSITIVE SCROLL on the cursor to force the use of a temporary copy of the result table.
- It can derive the result table rows as they are needed during the execution of later FETCH statements.

If the result table is not read-only, DB2 uses the latter method. If the result table is read-only, either method could be used. The results produced by these two methods could differ in the following respects:

When a temporary copy of the result table is used: An error can occur that would otherwise not occur until some later FETCH statement. insert operations that are executed while the cursor is open cannot affect the result table once all the rows have been materialized in the temporary copy of the result table. For a scrollable insensitive cursor, update and delete operations that are executed while the cursor is open cannot affect the result table. For a scrollable sensitive static cursor, update and delete operations can affect the result table if the rows are subsequently fetched with sensitive FETCH statements.

When a temporary copy of the result table is not used: Insert, update, and delete operations that are executed while the cursor is open can affect the result table. The effect of such operations is not always predictable.

For example, if cursor C is positioned on a row of its result table defined as SELECT * FROM T, and you insert a row into T, the effect of that insert on the result table is not predictable because its rows are not ordered. A later FETCH C might or might not retrieve the new row of T. To avoid these changes, you can specify INSENSITIVE SCROLL for the cursor to force the use of a temporary copy of the result table.

Parameter marker replacement: Before the OPEN statement is executed, each parameter marker in the query is effectively replaced by its corresponding host variable. The replacement is an assignment operation in which the source is the value of the host variable and the target is a variable within DB2. The assignment rules are those described for assignment to a column in "Assignment and comparison" on page 121. For a typed parameter marker, the attributes of the target variable are those specified by the CAST specification. For an untyped parameter marker, the attributes of the target variable are determined according to the context of the parameter marker. For the rules that affect parameter markers, see Parameter markers.

Let V denote a host variable that corresponds to parameter marker P. The value of V is assigned to the target variable for P in accordance with the rules for assigning a value to a column:

- V must be compatible with the target.
- If V is a string, its length (excluding trailing blanks) must not be greater than the length attribute of the target.
- If V is a number, the absolute value of its integral part must not be greater than the maximum absolute value of the integral part of the target.
- If the attributes of V are not identical to the attributes of the target, the value is converted to conform to the attributes of the target.
- If the target cannot contain nulls, V must not be null.

When the SELECT statement of the cursor is evaluated, each parameter marker in the statement is effectively replaced by the value of its corresponding host variable. For example, if V is CHAR(6) and the target is CHAR(8), the value used in place of P is the value of V padded on the right with two blanks. For more on the process of replacement, see Parameter marker replacement.

Considerations for scrollable cursors: Following an OPEN *cursor* statement, a GET DIAGNOSTICS statement can be used to get the attributes of the cursor such as the following information (for more information, see "GET DIAGNOSTICS" on page 1679):

- DB2_SQL_ATTR_CURSOR _HOLD. Whether the cursor was defined with the WITH HOLD attribute.
- DB2_SQL_ATTR_CURSOR_SCROLLABLE. Scrollability of the cursor.
- DB2_SQL_ATTR_CURSOR_SENSITIVITY. Effective sensitivity of the cursor.

  The sensitivity information can be used by applications (such as an ODBC driver) to determine what type of FETCH (INSENSITIVE or SENSITIVE) to issue for a cursor defined as ASENSITIVE.
- DB2_SQL_ATTR_CURSOR_ROWSET. Whether the cursor can be used to access rowsets.
- DB2_SQL_ATTR_CURSOR_TYPE. Whether a cursor type is forward-only, static, or dynamic.
- The scrollability of the cursor is in SQLWARN1.
- The sensitivity of the cursor is in SQLWARN4.

- The effective capability of the cursor is in SQLWARN5.

Number of rows inserted: SQL data change statements and routines that modify SQL data embedded in the cursor definition are completely executed, and the result table is stored in a temporary table when the cursor opens. If statement execution is successful, the SQLERRD(3) field contains the sum of the number of rows that qualified for insert, update, and delete operations. If an error occurs during execution of an OPEN statement that involves a cursor that contains a data change statement within a fullselect, the results of that data change statement are rolled back.

Materialization of the rows of the result table and NEXT VALUE expressions: If the rows of the result table of a cursor are materialized when the cursor is opened and the SELECT statement of the cursor contains NEXT VALUE expressions, the expressions are processed when the cursor is opened. Otherwise, the NEXT VALUE expressions are evaluated as the rows of the result table are retrieved.

Opening the same cursor multiple times: A cursor in an SQL procedure that is declared as WITH RETURN TO CLIENT can be opened even when a cursor with the same name is already in the open state. In this case, the existing open cursor becomes a result set cursor and is no longer accessible by its cursor name. A new cursor is opened and becomes accessible by the cursor name. Closing the new cursor does not make the cursor that was previously accessible by that name accessible by the cursor name again. Cursors that become result set cursors in this way cannot be accessed at the server and can be processed only at the client.

Examples

Example 1: Execute an OPEN statement, which places the cursor at the beginning of the rows to be fetched.

```
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT DEPTNO, DEPTNAME, MGRNO FROM DSN8B10.DEPT
  WHERE ADMRDEPT = 'A00';
EXEC SQL OPEN C1;
 DO WHILE (SQLCODE = 0);
  EXEC SQL FETCH C1 INTO :DNUM, :DNAME, :MNUM;
END;
 EXEC SQL CLOSE C1;
```

Example 2: Suppose that the following array type, array variable, and table have been defined.

```
CREATE TYPE INTARRAY AS INTEGER ARRAY[100];
CREATE TYPE STRINGARRAY AS VARCHAR(10) ARRAY[100];
CREATE TABLE T1 (COL1 CHAR(10), COL2 INT);
```

Use an array variable as input for a dynamic SQL statement. The dynamic statement references an array element in the array variable. The dynamic statement contains two parameter markers, one for the array variable and one for the index of the array element. The OPEN statement provides two input values in the USING clause: the array variable, and a variable that contains the index for the array element.

```
CREATE PROCEDURE PROCESSPERSONS (OUT WITHO STRINGARRAY, INOUT INT0 INT)
BEGIN
 DECLARE INTA INTARRAY;
 DECLARE INTB INTARRAY;
 DECLARE INTV INTEGER;
 DECLARE STMT CHAR(100);
 DECLARE C2 CURSOR FOR S1;
```

```
--
-- Initialize the array
--
 SET INTA = ARRAY[1,INTEGER(2),3+0,4,5,6] ;
--
-- Use dynamic SQL with an array parameter marker and a parameter marker
-- containing the index to retrieve the value from the array parameter.
-- The array is referenced in a predicate.
--
 SET STMT = 'SELECT COL1 FROM T1 WHERE COL2 = CAST(? AS INTARRAY)[?]';
 PREPARE S1 FROM STMT;
 OPEN C2 USING INTA, INTV; -- Input: INTA is an array, and INTV is the
                           -- index for the array element
 FETCH C2 INTO INTB ;      -- Output: INTB is an array variable
...
 CLOSE C2;
...
END
```

PREPARE

The PREPARE statement creates an executable SQL statement from a string form of the statement. The character-string form is called a *statement string*. The executable form is called a *prepared statement*.

Invocation

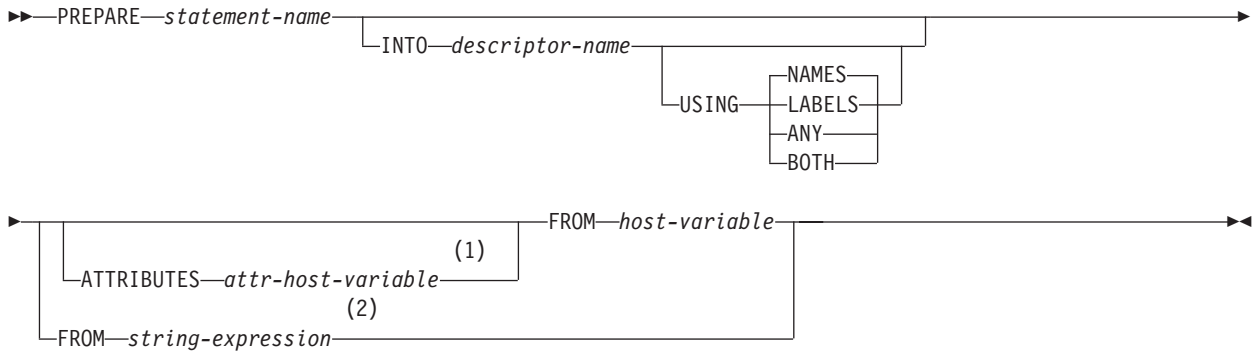
This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

The authorization rules are those defined for the dynamic preparation of the SQL statement specified by the PREPARE statement. For example, see Chapter 4, "Queries," on page 761 for the authorization rules that apply when a SELECT statement is prepared.

The statement that is prepared using only the EXPLAIN privilege cannot be executed, and only the descriptive information can be obtained for that statement.

Syntax

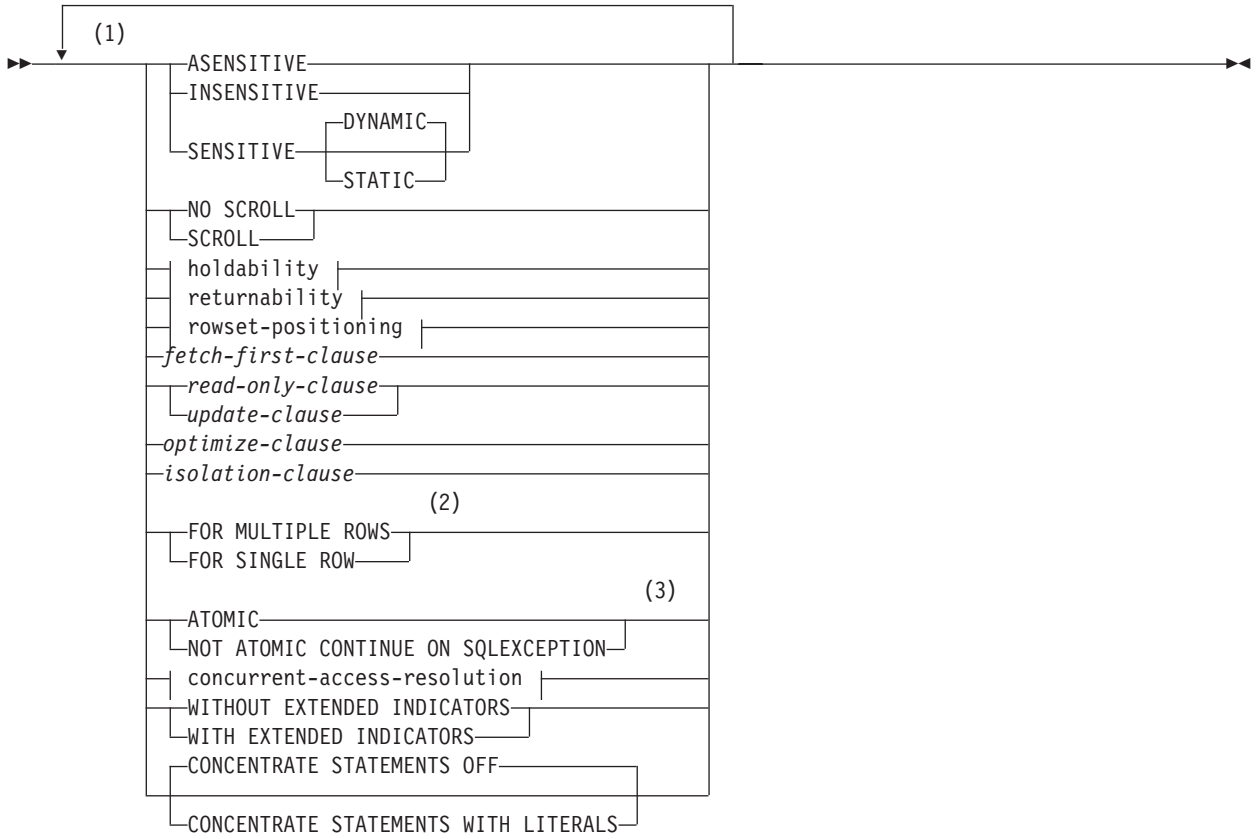


Notes:

1 *attr-host-variable* must be a string host variable and the content must conform to the rules for attribute-string. The ATTRIBUTES clause can only be specified before *host-variable*.

2 *string-expression* is only supported for PLI.

attribute-string



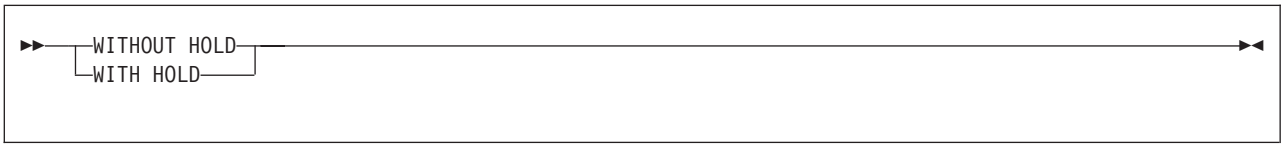
Notes:

1 The same clause must not be specified more than one time. If the options are not specified, their defaults are whatever was specified for the corresponding option in an associated statement.

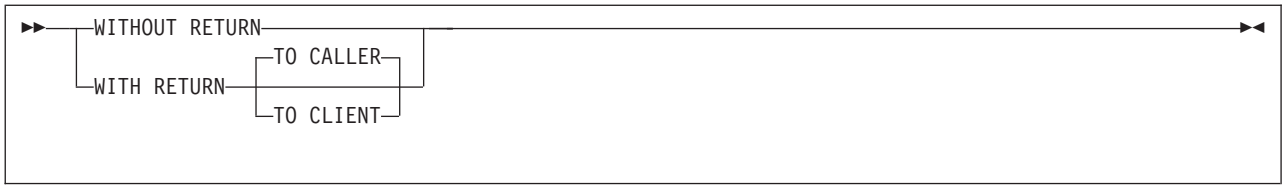
2 The FOR SINGLE ROW or FOR MULTIPLE ROWS clause must only be specified for an INSERT or a MERGE statement.

3 The ATOMIC or NOT ATOMIC CONTINUE ON SQLEXCEPTION clause must only be specified for an INSERT statement.

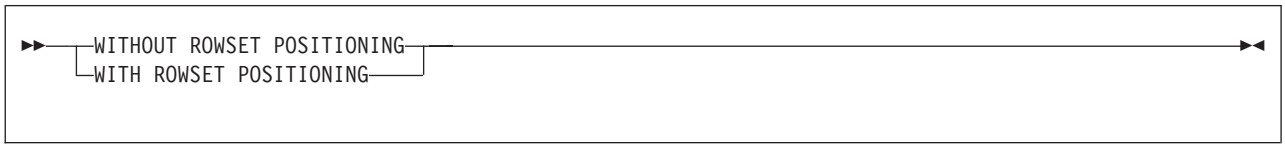
holdability:



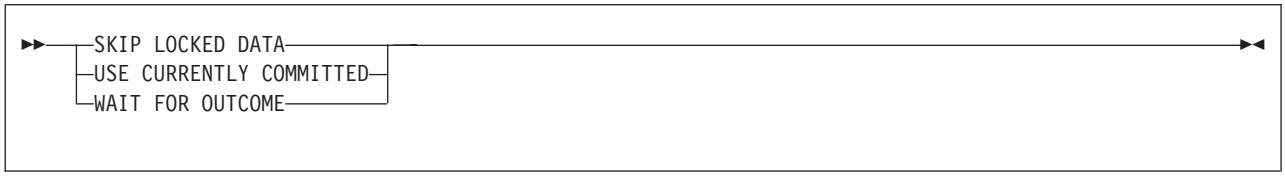
returnability:



rowset-positioning:



concurrent-access-resolution



Description

statement-name
Names the prepared statement. If the name identifies an existing prepared statement, that prepared statement is destroyed. The name must not identify a prepared statement that is the SELECT statement of an open cursor.

INTO
If you use INTO, and the PREPARE statement is successfully executed, information about the prepared statement is placed in the SQLDA specified by the descriptor name. Thus, the PREPARE statement:

```
EXEC SQL PREPARE S1 INTO :SQLDA FROM :V1;
```

is equivalent to:

```
EXEC SQL PREPARE S1 FROM :V1;
EXEC SQL DESCRIBE S1 INTO :SQLDA;
```

descriptor-name
Identifies the SQLDA. For languages other than REXX, SQLN must be set to indicate the number of SQLVAR occurrences. See “DESCRIBE” on page 1590 and “SQL descriptor area (SQLDA)” on page 2079 for information about how to determine the number of SQLVAR occurrences to use and for an explanation of the information that is placed in the SQLDA.

See “Identifying an SQLDA in C or C++” on page 2099 for how to represent *descriptor-name* in C.

USING
Indicates what value to assign to each SQLNAME variable in the SQLDA when INTO is used. If the requested value does not exist, SQLNAME is set to length 0.

NAMES
Assigns the name of the column. This is the default.

LABELS
Assigns the label of the column. (Column labels are defined by the LABEL statement.)

ANY
Assigns the column label, and, if the column has no label, the column name.

BOTH
Assigns both the label and name of the column. In this case, two or three occurrences of SQLVAR per column, depending on whether the result table contains distinct types, are needed to accommodate the additional information. To specify this expansion of the SQLVAR array, set SQLN to 2×*n* or 3×*n*, where *n* is the number of columns in the object being described. For each of the columns, the first *n* occurrences of SQLVAR, which are the base SQLVAR entries, contain the column names. Either the second or third *n* occurrences of SQLVAR, which are the extended SQLVAR entries, contain the column labels. If there are no distinct types, the labels are returned in the second set of SQLVAR entries. Otherwise, the labels are returned in the third set of SQLVAR entries.

A REXX SQLDA does not include the SQLN field, so you do not need to set SQLN for REXX programs.

ATTRIBUTES *attr-host-variable*
Specifies the attributes that are in effect if a corresponding attribute has not been specified as part of the associated statement. If attributes are specified as part of the associated statement, they are used instead of the corresponding attributes specified on the PREPARE statement. In turn, if attributes are specified in the PREPARE of a SELECT statement, they are used instead of the corresponding attributes specified on a DECLARE CURSOR statement.

attr-host-variable must identify a host variable that is described in the program in accordance with the rules for declaring string variables. *attr-host-variable* must be a string variable (either fixed-length or varying-length) that has a length attribute that does not exceed 32758 bytes. Leading and trailing blanks are removed from the value of the host variable. The host variable must contain a valid *attribute-string*.

An indicator variable can be used to indicate whether or not attributes are actually provided on the PREPARE statement. Thus, applications can use the same PREPARE statement regardless of whether attributes need to be specified or not.

The options that can be specified as part of the *attribute-string* are as follows:

ASENSITIVE, INSENSITIVE, SENSITIVE STATIC, or SENSITIVE DYNAMIC
Specifies the sensitivity of the cursor to inserts, updates, or deletes that made to the rows underlying the result table. The sensitivity of the cursor determines whether DB2 can materialize the rows of the result into a temporary table. The default is ASENSITIVE.

ASENSITIVE
Specifies that the cursor should be as sensitive as possible. A cursor that defined as ASENSITIVE will be either insensitive or sensitive

36. The scrollability and sensitivity of the cursor are independent and do not have to be specified together. Thus, the cursor might be defined as SCROLL INSENSITIVE, but the PREPARE statement might specify SENSITIVE STATIC as an override for the sensitivity.

dynamic; it will not be sensitive static. For information about how the effective sensitivity of the cursor is returned to the application with the GET DIAGNOSTICS statement or in the SQLCA, see "OPEN" on page 1775.

INSENSITIVE
Specifies that the cursor does not have sensitivity to inserts, updates, or deletes that are made to the rows underlying the result table. As a result, the size of the result table, the order of the rows, and the values for each row do not change after the cursor is opened. In addition, the cursor is read-only. The SELECT statement or *attribute-string* of the PREPARE statement cannot contain a FOR UPDATE clause, and the cursor cannot be used for positioned updates or deletes.

SENSITIVE
Specifies that the cursor has sensitivity to changes made to the database after the result table is materialized. The cursor is always sensitive to positioned updates and deletes that are made using the same cursor. However, the *select-statement* of the cursor must not contain an SQL data change statement if the cursor is defined as either SENSITIVE DYNAMIC or SENSITIVE STATIC. When the current value of a row no longer satisfies the *select-statement* or *statement-name*, that row is no longer visible through the cursor. When a row of the result table is deleted from the underlying base table, the row is no longer visible through the cursor.

In addition, the cursor has sensitivity to changes made to values outside the cursor (that is, by other cursors or committed changes by other application processes). If DB2 can not make changes made outside the cursor visible to the cursor, an error is issued at OPEN CURSOR. Whether the cursor is sensitive to changes made outside this cursor depends on whether DYNAMIC or STATIC is in effect for the cursor and whether SENSITIVE or INSENSITIVE FETCH statements are used.

Whether the cursor is sensitive to newly inserted rows depends on whether DYNAMIC or STATIC is in effect for the cursor. The default is DYNAMIC.

DYNAMIC
Specifies that the result table of the cursor is dynamic in that the size of the result table can change after the cursor is opened as rows are inserted into or deleted from the underlying table, and the order of the rows can change. Inserts, deletes, and updates that are made by the same application process are immediately visible. Inserts, deletes, and updates that are made by other application processes are visible after they are committed.

All FETCH statements for sensitive dynamic cursors are sensitive to changes made by this cursor, changes made by other cursors in the same application process, and committed changes made by other application processes.

If a SENSITIVE DYNAMIC cursor is not possible, an error is returned, an error is returned. The FETCH FIRST *n* ROWS ONLY clause must not be specified for the outermost fullselect for a sensitive dynamic cursor.

STATIC
Specifies that the order of the rows and size of the result table is

static. The size of the result table does not grow after the cursor is opened and the rows are materialized. The order of the rows is established as the result table is materialized. Rows that are inserted into the underlying table are not added to the result table of the cursor regardless of how the rows were inserted. Rows in the result table do not move if columns in the ORDER BY clause are updated in rows that have already been materialized.

Whether the changes that are made outside the cursor are visible to the cursor depends on the type of FETCH that is used with a SENSITIVE STATIC cursor. For more information, see Considerations for FETCH statements used with a sensitive static cursor.

Using a function that is not deterministic (built-in or user-defined) in the WHERE clause of *select-statement* or *statement-name* of a SENSITIVE STATIC cursor can cause misleading results. This occurs because DB2 constructs a temporary result table and retrieves rows from this table for INSENSITIVE FETCH statements. When DB2 processes a SENSITIVE FETCH statement, rows are fetched from the underlying table and predicates are re-evaluated if they contain non-correlated subqueries. Using a function that is not deterministic can yield a different result for the re-evaluated query causing the row to no longer be considered a match.

If SENSITIVE STATIC is specified and a sensitive static cursor is not possible, then an error is returned.

If ASENSITIVE, INSENSITIVE, SENSITIVE DYNAMIC, or SENSITIVE STATIC is specified as part of the ATTRIBUTES clause, SCROLL must be specified.

SCROLL or NO SCROLL
Specifies whether the cursor is scrollable.

SCROLL
Specifies that the cursor is scrollable.

NO SCROLL
Specifies that the cursor is not scrollable.

WITHOUT RETURN or WITH RETURN
Specifies whether the result table of the cursor is intended to be used as a result set that will be returned from a procedure. If *statement-name* is specified, the default is the corresponding prepare attribute of the statement. Otherwise, the default is **WITHOUT RETURN**.

WITHOUT RETURN
Specifies that the result table of the cursor is not intended to be used as a result set that will be returned from a procedure.

WITH RETURN
Specifies that the result table of the cursor is intended to be used as a result set that will be returned from a procedure. WITH RETURN is relevant only if the PREPARE statement is contained within the source code for a procedure. In other cases, the precompiler might accept the clause, but it has not effect.

When a cursor that is declared using the WITH RETURN TO CALLER clause remains open at the end of a program or routine, that cursor defines a result set from the program or routine. Use the CLOSE

statement to close a cursor that is not intended to be a result set from the program or routine. Although DB2 will automatically close any cursors that are not declared using with a WITH RETURN clause, the use of the CLOSE statement is recommended to increase the portability of applications.

For non-scrollable cursors, the result set consists of all rows from the current cursor position to the end of the result table. For scrollable cursors, the result set consists of all rows of the result table.

TO CALLER
Specifies that the cursor can return a result set to the caller of the procedure. The caller is the program or routine that executed the SQL CALL statement that invokes the procedure that contains the PREPARE statement. For example, if the caller is a procedure, the result set, is returned to the procedure. If the caller is a client application, the result set is returned to the client application.

If the statement is contained within the source code for a procedure, WITH RETURN TO CALLER specifies that the cursor can be used as a result set cursor. A result set cursor is used when the result table of a cursor is to be returned from a procedure. Specifying TO CALLER is optional.

In other cases, the clause is ignored and the cursor cannot be used as a result set cursor.

TO CLIENT
Specifies that the cursor can return a result set to the client application. This cursor is invisible to any intermediate nested procedures. If a function or trigger calls the procedure (either directly or indirectly), the result set cannot be returned to the client and the cursor will be closed after the procedure finishes.

rowset-positioning
Specifies whether rows of data can be accessed as a rowset on a single FETCH statement for this cursor.

WITHOUT ROWSET POSITIONING
Specifies that the cursor can only be used with row positioned FETCH statements.

WITH ROWSET POSITIONING
Specifies that this cursor can be used with rowset positioned or row positioned FETCH statements

fetch-first-clause
Limits the number of rows that can be fetched. It improves the performance of queries with potentially large result sets when only a limited number of rows are needed. If the clause is specified, the number of rows retrieved will not exceed *n*, where *n* is the value of the integer. An attempt to fetch n+1 rows is handled the same way as normal end of date. The value of integer must be positive and non-zero. The default is 1.

If the OPTIMIZE FOR clause is not specified, a default of OPTIMIZE FOR *integer* ROWS is assumed. If both the FETCH FIRST and OPTIMIZE FOR clauses are specified, the lower of the integer values from these clauses is used to influence optimization and the communications buffer size.

The FETCH FIRST clause must not be specified for the outermost fullselect for a sensitive dynamic cursor.

read-only-clause
Declares that the result table is read-only and therefore the cursor cannot be referred to in positioned UPDATE and DELETE statements.

update-clause
Identifies the columns that can updated in a later positioned UPDATE statement. Each column must be unqualified and must identify a column of the table or view identified in the first FROM clause of the fullselect. The clause must not be specified if the result table of the fullselect is read-only. The clause must also not be specified if a created temporary table is referenced in the first FROM clause of the select-statement.

If the clause is specified without a list of columns, the columns that can be updated include all the updatable columns of the table or view that is identified in the first FROM clause of the fullselect.

optimize-clause
Requests special optimization of the *select-statement*. If the clause is omitted, optimization is based on the assumption that all rows of the result table will be retrieved. If the clause is specified, optimization is based on the assumption that the number of rows retrieved will not exceed *n*, where *n* is the value of the integer. The clause does not limit the number of rows that can be fetched or affect the result in any way other than performance.

isolation-clause
Specifies the isolation level at which the select statement is executed. See "isolation-clause" on page 827.

concurrent-access-resolution
Specifies the type concurrent access resolution to use for the select statement. Each clause in *concurrent-access-resolution* can only be specified one time. Only one of the clauses can be specified for each PREPARE statement. If none of the clauses is specified, the locking semantic depends on other attributes of the statement.

SKIP LOCKED DATA
Specifies to skip data on which incompatible locks are held by other transactions. See "SKIP LOCKED DATA" on page 830.

USE CURRENTLY COMMITTED
Specifies that DB2 can use the currently committed version of the data when the data is in the process of being updated. USE CURRENTLY COMMITTED only applies in the following cases:

- The table that is being accessed is defined in a universal table space
- The access is for a select-statement with an isolation level of cursor stability (CS) or read stability (RS) specified in the isolation-clause:
  - When a read transaction accesses a record that is locked by an insert transaction, both ISOLATION(CS) and ISOLATION(RS) are applicable.
  - When a read transaction accesses a record that is locked by a delete transaction only ISOLATION(CS) is applicable and only when CURRENTDATA(NO) is in effect.

USE CURRENTLY COMMITTED is ignored if used in any other context.

When this clause is specified, the setting of the subsystem parameter EVALUNC applies. If the row qualifies, this clause determines if the row is accessed or skipped.

When this clause is specified and the subsystem parameter SKIPUNCI is in effect, PREPARE uses the specification of this clause. See the Notes section for more information.

When this clause is specified and XML data that does not support multiple XML versions is being selected, DB2 cannot determine whether the data has been committed. In this case, DB2 uses WAIT FOR OUTCOME behavior when accessing the data.

WAIT FOR OUTCOME
Specifies that DB2 waits for the commit or rollback when encountering data that is in the process of being updated or deleted. Rows that are in the process of being inserted are not skipped.

FOR MULTIPLE ROWS or FOR SINGLE ROW
Specifies if a variable number of rows will be provided for a dynamic INSERT or MERGE statement.

FOR MULTIPLE ROWS
Specifies that multiple rows can be provided with host variable arrays on an EXECUTE statement for the statement that is being prepared. FOR MULTIPLE ROWS must only be specified for an INSERT or a MERGE statement.

FOR SINGLE ROW
Specifies that multiple rows must not be provided with host variable arrays on an EXECUTE statement for the statement that is being prepared. FOR SINGLE ROW must only be specified for an INSERT or a MERGE statement.

ATOMIC or NOT ATOMIC CONTINUE ON SQLEXCEPTION
Specifies if all rows are inserted as an atomic operation. This clause can only be specified for dynamic INSERT statements.

ATOMIC
Specifies that if the insert for any row fails, all changes that are made to the database by any of the inserts, including changes that are made by successful inserts, are undone. This is the default.

NOT ATOMIC CONTINUE ON SQLEXCEPTION
Specifies that, regardless of the failure of any particular insert of a row, the INSERT statement will not undo any changes that are made to the database by the successful inserts of other rows, and inserting will be attempted for subsequent rows. However, the minimum level of atomicity is at least that of a single insert (that is, it is not possible for a partial insert operation to complete), including any triggers that might have been activated as a result of the INSERT statement.

This clause must not be specified if the INSERT statement is contained within a SELECT statement.

For preparing the MERGE statement, atomicity is specified only on the MERGE statement itself.

WITHOUT EXTENDED INDICATORS or WITH EXTENDED INDICATORS
Specifies whether the values that are provided for indicator variables during the execution of the statement follow standard SQL semantics for indicating NULL values, or if the values can use extended indicator variables to indicate a DEFAULT or UNASSIGNED value. WITHOUT EXTENDED INDICATORS is the default.

CONCENTRATE STATEMENTS OFF or CONCENTRATE STATEMENTS WITH LITERALS
Specifies whether a dynamic SQL statement that specifies literal constants will be cached as a separate unique statement entry in the dynamic statement cache instead of sharing an existing statement in the cache. Dynamic SQL statements are eligible to share an existing statement in the cache if the new statement meets all of the conditions for sharing a cached version of the same dynamic statement except that the new statement specifies one or more literal constants that are different than the cached statement.

CONCENTRATE STATEMENTS OFF
Specifies that the dynamic SQL statement that specifies literal constants will be cached as a unique statement entry if it specifies one or more constants that are different than the cached version of the same dynamic statement. **CONCENTRATE STATEMENTS OFF** is the default dynamic statement caching behavior.

CONCENTRATE STATEMENTS WITH LITERALS
Specifies that the dynamic SQL statement that specifies literal constants will share a cached version of the same dynamic statement that is also prepared using the **CONCENTRATE STATEMENTS WITH LITERALS** option if the new dynamic statement meets all of the conditions for sharing the cached statement and the constants that are specified can be reused in place of the constants in the cached statement.

FROM
Specifies the statement string. The statement string is the value of the specified *string-expression* or the identified *host-variable*.

host-variable
Must identify a host variable that is described in the application program in accordance with the rules for declaring string variables. If the source string is over 32KB in length, the *host-variable* must be a CLOB or DBCLOB variable. The maximum source string length is 2MB although the host variable can be declared larger than 2MB. An indicator variable must not be specified. In PL/I, COBOL and Assembler language, the host variable must be a varying-length string variable. In C, the host variable must not be a NUL-terminated string. In SQL PL, an SQL variable is used in place of a host variable and the value must not be null.

string-expression
string-expression is any PL/I expression that yields a string. *string-expression* cannot be preceded by a colon. Variables that are within *string-expression* that include operators or functions should not be preceded by a colon. When *string-expression* is specified, the precompiler-generated structures for *string-expression* use an EBCDIC CCSID and an informational message is returned.

Notes

Rules for statement strings: The value of the specified *statement-name* is called the *statement string*. The statement string must be one of the following SQL statements:

ALLOCATE CURSOR	RELEASE SAVEPOINT
ALTER	RENAME
ASSOCIATE LOCATORS	REVOKE
COMMENT	ROLLBACK
COMMIT	SAVEPOINT
CREATE	*select-statement*
DECLARE GLOBAL TEMPORARY TABLE	SET CURRENT DEGREE
DELETE	SET CURRENT DEBUG MODE
DROP	SET CURRENT DECFLOAT ROUNDING MODE
EXPLAIN	SET CURRENT LOCALE LC_CTYPE
FREE LOCATOR	SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION
GRANT	SET CURRENT OPTIMIZATION HINT
HOLD LOCATOR	SET CURRENT PRECISION
INSERT	SET CURRENT QUERY ACCELERATION
LABEL	SET CURRENT REFRESH AGE
LOCK TABLE	SET CURRENT ROUTINE VERSION
MERGE	SET CURRENT RULES
REFRESH TABLE	SET CURRENT SQLID
	SET ENCRYPTION PASSWORD
	SET PATH
	SET SCHEMA
	SET SESSION TIME ZONE
	SIGNAL
	TRUNCATE
	UPDATE

The statement string must not:

- Begin with EXEC SQL
- End with END-EXEC or a semicolon
- Include references to variables

Parameter markers: Although a statement string cannot include references to host variables, it can include *parameter markers*. The parameter markers are replaced by the values of host variables when the prepared statement is executed. A parameter marker is a question mark (?) that appears where a host variable could appear if the statement string were a static SQL statement. For an explanation of how parameter markers are replaced by values, see the EXECUTE statement, "OPEN" on page 1775, and *DB2 Application Programming and SQL Guide*.

The two types of parameter markers are typed and untyped:

Typed parameter marker
A parameter marker that is specified with its target data type. A typed parameter marker has the general form:

```
CAST(? AS data-type)
```

This invocation of a CAST specification is a "promise" that the data type of the parameter at run time will be of the data type that is specified or some data type that is assignable to the specified data type. For example, in the following UPDATE statement, the value of the argument of the TRANSLATE function will be provided at run time:

```
UPDATE EMPLOYEE
  SET LASTNAME = TRANSLATE(CAST(? AS VARCHAR(12)))
 WHERE EMPNO = ?
```

The data type of the value that is provided for the TRANSLATE function will either be VARCHAR(12), or some data type that can be converted to VARCHAR(12). For more information, refer to "Assignment and comparison" on page 121.

Untyped parameter marker
A parameter marker that is specified without its target data type. An untyped parameter marker has the form of a single question mark. The context in which the parameter marker appears determines its data type. For example, in the above UPDATE statement, the data type of the untyped parameter marker in the predicate is the same as the data type of the EMPNO column.

Typed parameter markers can be used in dynamic SQL statements wherever a host variable is supported and the data type is based on the promise made in the CAST specification.

Untyped parameters markers can be used in dynamic SQL statements in selected locations where host variables are supported. Table 147, Table 148 on page 1794, Table 149 on page 1795, and Table 150 on page 1796 show these locations and the resulting data type of the parameter. The tables group the locations into expressions, predicates, functions, and other statements to help show where untyped parameter markers are allowed.

Table 147. Untyped parameter marker usage in expressions (including select list, CASE, and VALUES)

Location of untyped parameter marker	Data type (or error if not supported)
Alone in a select list. For example: `SELECT ?`	Error
Both operands of a single arithmetic operator, after considering operator precedence and the order of operation rules. Includes cases such as: `? + ? + 10`	DECFLOAT(34)
One operand of a single operator in an arithmetic expression (except datetime arithmetic expressions). Includes cases such as: `? + ? * 10`	The data type of the other operand
Any operand of a datetime expression. For example: `'timecol + ?' or '? - datecol'`	Error
A labeled duration in a datetime expression with a type unit other than SECONDS (the portion of a labeled duration that indicates the type of units cannot be a parameter marker).	DECIMAL(15,0)
A labeled duration in a datetime expression with a type unit of SECONDS (the portion of a labeled duration that indicates the type of units cannot be a parameter marker).	DECIMAL(27,12)
Both operands of a CONCAT operator	Error

Table 147. Untyped parameter marker usage in expressions (including select list, CASE, and VALUES) (continued)

Location of untyped parameter marker	Data type (or error if not supported)
One operand of a CONCAT operator when the other operand is any character data type except CLOB	If the other operand is CHAR(*n*) or VARCHAR(*n*), where n is less than 128, the data type is VARCHAR(254 - *n*). In all other cases, the data type is VARCHAR(254).
One operand of a CONCAT operator when the other operand is any graphic data type except DBCLOB	If the other operand is GRAPHIC(*n*) or VARGRAPHIC(*n*), where n is less than 64, the data type is VARGRAPHIC(127 - *n*). In all other cases, the data type is VARGRAPHIC(127).
One operand of a CONCAT operator when the other operand is any binary type except BLOB	If the other operand is BINARY(n) or VARBINARY(n) where n is less than 128, the data type is VARBINARY(255-n). In all other cases, the data type is VARBINARY(255)
One operand of a CONCAT operator when the other operand is a LOB string	The data type of the other operand (the LOB string)
The *expression* following the CASE keyword in a simple CASE expression	Error
Any or all *expressions* following the WHEN keyword in a simple CASE expression	The result of applying the "Rules for result data types" on page 144 to the expression following CASE and the expressions following WHEN that are not untyped parameter markers
A *result-expression* in any CASE expression when all the other *result-expression*s are either NULL or untyped parameter markers.	Error
A *result-expression* in any CASE expression when at least one other *result-expression* is neither NULL nor an untyped parameter marker.	The result of applying the "Rules for result data types" on page 144 to all the *result-expression*s that are not NULL or untyped parameter markers
Alone as a *column-expression* in a single-row VALUES clause that is not within an INSERT statement or the VALUES clause of in insert operation of a MERGE statement	Error
Alone as a *column-expression* in a single-row VALUES clause within an INSERT statement	The data type of the column or, if the column is defined as a distinct type, the source data type of the distinct type
Alone as a *column-expression* in a *values-single-row* or *values-multiple-row* clause of *source-table* for a MERGE statement	The data type of the column of the source-table, or if the data type is a distinct type, the source data type of the distinct type. The column of the source-table must be referenced elsewhere in the MERGE statement such that its data type can be determined from the context in which it is used, and all such references must resolve to the same data type.
Alone as a *column-expression* in the VALUES clause of an insert operation of a MERGE statement	The data type of the column or, if the column is defined as a distinct type, the source data type of the distinct type
Alone as a *column-expression* on the right side of *assignment-clause* for an update operation of a MERGE statement	The data type of the column or, if the column is defined as a distinct type, the source data type of the distinct type

Table 147. Untyped parameter marker usage in expressions (including select list, CASE, and VALUES) (continued)

Location of untyped parameter marker	Data type (or error if not supported)
Alone as a *column-expression* on the right side of a SET clause in an UPDATE statement	The data type of the column or, if the column is defined as a distinct type, the source data type of the distinct type

Table 148. Untyped parameter marker usage in predicates

Location of untyped parameter marker	Data type (or error if not supported)
Both operands of a comparison operator	Error
One operand of a comparison operator when the other operand is not an untyped parameter marker	The data type of the other operand. If the operand has a datetime data type, the result of DESCRIBE INPUT will show the data type as CHAR(255) although DB2 uses the datetime data type in any comparisons.
All the operands of a BETWEEN predicate	Error
Two operands of a BETWEEN predicate (either the first and second, or the first and third)	The data type of the operand that is not a parameter marker
Only one operand of a BETWEEN predicate	The result of applying the "Rules for result data types" on page 144 on the other operands that are not parameter markers
All the operands of an IN predicate, for example, `? IN (?,?,?)`	Error
The first and second operands of an IN predicate, for example, `? IN (?,A,B)`	The result of applying the "Rules for result data types" on page 144 on the operands in the IN list that are not parameter markers
The first operand of an IN predicate and zero or more operands of the IN list except for the first operand of the IN list, for example, `? IN (A,?,B,?)`	The result of applying the "Rules for result data types" on page 144 on the operands in the IN list that are not parameter markers
The first operand of an IN predicate when the right side is a fullselect of fullselect, for example, `? IN (fullselect)`	The data type of the selected column
Any or all operands of the IN list of the IN predicate and the first operand of the IN predicate is not an untyped parameter marker, for example, `A IN (?,A,?)`	The data type of the first operand (the operand on the left side of the IN list)
All the operands of a LIKE predicate	The first and second operands (*match-expression* and *pattern-expression*) are VARCHAR(4000). The third operand (*escape-expression*) is VARCHAR(1).
The first operand of a LIKE predicate (the *match-expression*) when at least one other operand (the *pattern-expression* or *escape-expression*) is not an untyped parameter marker.	VARCHAR(4000), VARGRAPHIC(2000), or VARBINARY(4000), depending on the data type of the first operand that is not an untyped parameter marker

Table 148. Untyped parameter marker usage in predicates (continued)

Location of untyped parameter marker	Data type (or error if not supported)
The second operand of a LIKE predicate (the *pattern-expression*) when at least one other operand (the *match-expression* or *escape-expression*) is not an untyped parameter marker. When the pattern specified in a LIKE predicate is a parameter marker and a fixed-length character host variable is used to replace the parameter marker, specify a value for the host variable that is the correct length. If you do not specify the correct length, the select does not return the intended results.	VARCHAR(4000), VARGRAPHIC(2000), or VARBINARY(4000), depending on the data type of the first operand that in not an untyped parameter marker.
The third operand of a LIKE predicate (the *escape-expression*) when at least one other operand (the *match-expression* or *pattern-expression*) is not an untyped parameter marker	CHAR(1), GRAPHIC(1), or BINARY(1), depending on the data type of the first operand that in not an untyped parameter marker
Operand of a NULL predicate	Error

Table 149. Untyped parameter marker usage in functions

Location of untyped parameter marker	Data type (or error if not supported)
All arguments of COALESCE or NULLIF	Error
Any argument of COALESCE or NULLIF when at least one other argument is not an untyped parameter marker	The result of applying the "Rules for result data types" on page 144 on the arguments that are not untyped parameter markers, the data type of the other argument
First argument of COLLATION_KEY	VARGRAPHIC(2000)
Second argument of COLLATION_KEY	VARCHAR(255)
First argument of LOWER	VARCHAR(4000)
Second argument of LOWER	VARCHAR(255)
Any argument other than the first argument of MAX	The data type of the corresponding parameter in the function instance
Any argument other than the first argument of MIN	The data type of the corresponding parameter in the function instance
Both arguments of POSSTR or POSITION	VARCHAR(4000) for both arguments
One argument of POSSTR or POSITION when the other argument is a character data type	VARCHAR(4000)
One argument of POSSTR or POSITION when the other argument is a graphic data type	VARGRAPHIC(2000)
One argument of POSSTR or POSITION when the other argument is a BINARY or VARBINARY data type	VARBINARY(4000)
One argument of POSSTR or POSITION when the other argument is a BLOB	BLOB(4000)
First argument of SUBSTR or SUBSTRING	VARCHAR(4000)
Second or third argument of SUBSTR or SUBSTRING	INTEGER

Table 149. Untyped parameter marker usage in functions (continued)

Location of untyped parameter marker	Data type (or error if not supported)
One argument of TIMESTAMP	TIME
First argument of TIMESTAMP_FORMAT	VARCHAR(255)
First argument of TRANSLATE	Error
Second or third argument of TRANSLATE	VARCHAR(4000), VARGRAPHIC(2000), depending on whether the data type of the first argument is character or graphic
Fourth argument of TRANSLATE	VARCHAR(1) or VARGRAPHIC(1), depending on whether the data type of the first argument is character or graphic
Second argument of TRIM_ARRAY	BIGINT
array-index for *array-element-specification*	BIGINT
First argument of UPPER	VARCHAR(4000)
Second argument of UPPER	VARCHAR(255)
First argument of VARCHAR_FORMAT	TIMESTAMP WITHOUT TIME ZONE
Unary minus	DECFLOAT(34)
Unary plus	Error
The argument of any built-in scalar function (except those that are described in this table)	Error
The argument of a built-in aggregate function	Error
The argument of a user-defined scalar function, user-defined aggregate function, or user-defined table function	The data type of the corresponding parameter in the function instance

Table 150. Untyped parameter marker usage in statements

Location of untyped parameter marker	Data type (or error if not supported)
FOR *n* ROWS clause of an INSERT or MERGE statement	Integer
The value on the right side of a SET clause in an UPDATE statement or the UPDATE clause of the MERGE statement	The data type of the column of the source-table, or if the column is defined as a distinct type, the source data type of the distinct type. The column of the source-table must be referenced elsewhere in the MERGE statement such that its data type can be determined from the context in which it is used, and all such references must resolve to the same data type.
value, *value1*, or *value2* in a period specification for a table, or period clause for a data change statement if the target of the statement is a table	The data type of the columns of the period referenced in the period specification or period clause
value, *value1*, or *value2* in a period specification for a view	Error
value1 or *value2* in a period clause in a data change statement if the target of the statement is a view	Error

Considerations for FETCH statements used with a sensitive static cursor: Whether changes made outside the cursor are visible to the cursor depends on the type of FETCH that is used with a SENSITIVE STATIC cursor:

- A SENSITIVE FETCH is sensitive to all updates and deletes that are made by this cursor (including changes made by triggers) and committed updates and deletes by all other application processes because every fetched row is retrieved from the underlying base table and not a temporary table. This is the default type of FETCH statement for a SENSITIVE cursor.

  Changes that are made to the underlying data using this cursor result in an automatic refresh of the row. The changes that are made using this type of cursor can result in holes in the result table of the cursor. In addition, re-fetching rows (fetching rows that have already been retrieved) can result in holes in the result table. If a sensitive FETCH is issued to re-fetch a row and the row no longer qualifies for the search condition of the query, it results in a "delete hole" or an "update hole". In this case, no data is returned, and the cursor is left positioned on the hole.
- An INSENSITIVE FETCH is not sensitive to updates and deletes that are made outside this cursor; however, it is sensitive to all updates and deletes that are made by this cursor. Changes that made with triggers are not visible with an INSENSITIVE FETCH until the content of the rows are updated in the result table with a SENSITIVE FETCH statement. If an application does not want to be sensitive to changes that are made outside this cursor (that is, the application does not want to see changes made either with another cursor or by another application process), INSENSITIVE can be explicitly specified as part of the FETCH statement for a SENSITIVE STATIC cursor. This type of FETCH is useful for refreshing data in user data buffers. For more information, see INSENSITIVE.

Error checking: When a PREPARE statement is executed, the statement string is parsed and checked for errors. If the statement string is invalid, a prepared statement is not created and the error condition that prevents its creation is reported in the SQLCA.

In local and remote processing, the DEFER(PREPARE) and REOPT(ALWAYS)/ REOPT(ONCE) bind options can cause some SQL statements to receive “delayed” errors. For example, DESCRIBE, EXECUTE, and OPEN might receive an SQLCODE that normally occurs during PREPARE processing.

Reference and execution rules: Prepared statements can be referred to in the following kinds of statements, with the following restrictions shown:

In... The prepared statement...

DESCRIBE
has no restrictions

DECLARE CURSOR
must be SELECT when the cursor is opened

EXECUTE
must *not* be SELECT

A prepared statement can be executed many times. Indeed, if a prepared statement is not executed more than once and does not contain parameter markers, it is more efficient to use the EXECUTE IMMEDIATE statement rather than the PREPARE and EXECUTE statements.

Prepared statement persistence: All prepared statements created by a unit of work are destroyed when the unit of work is terminated, with the following exceptions:

- A SELECT statement whose cursor is declared with the option WITH HOLD persists over the execution of a commit operation if the cursor is open when the commit operation is executed.
- SELECT, INSERT, UPDATE, MERGE, and DELETE statements that are bound with KEEPDYNAMIC(YES) are kept past the commit operation if your system is enabled for dynamic statement caching, and none of the following are true:
  - SQL RELEASE has been issued for the site
  - Bind option DISCONNECT(AUTOMATIC) was used
  - Bind option DISCONNECT(CONDITIONAL) was used and there are no hold cursors for the site
- INSERT, UPDATE, MERGE, and DELETE statements that are bound with or use the RELEASE(DEALLOCATE) option and that reference a declared global temporary table are kept past commit operations unless one of the following statements is true:
  - The declared global temporary table is defined with the ON COMMIT DROP TABLE option.
  - The statement also references a DB2 base object (for example, a table or view), and one of the following statements is true:
    - The base object reference is for a DB2 catalog table.
    - At the commit point, DB2 determines that another DB2 thread is waiting for an X-lock on the base object's database descriptor (DBD).
    - The statement references an XML function or operation, and at the commit point DB2 determines that the base object DBD S-lock for the XML operation must be released.
    - At the commit point, DB2 determines that a base object DBD S-lock that is used by the statement must be released and cannot be maintained across the commit point.
  - DB2 determines that another DB2 thread is waiting for an X-lock on the DB2 package that contains the statement.

Scope of a statement name: The scope of a *statement-name* is the same as the scope of a *cursor-name*. See "DECLARE CURSOR" on page 1535 for more information about the scope of a *cursor-name*.

Preparation with PREPARE INTO and REOPT bind option: If bind option REOPT(ALWAYS) or REOPT(ONCE) is in effect, PREPARE INTO is equivalent to a PREPARE and a DESCRIBE being performed. If a statement has input variables, the DESCRIBE causes the statement to be prepared with default values, and the statement must be prepared again when it is opened or executed. When REOPT(ONCE) is in effect, the statement is always prepared twice even if there are no input variables. Therefore, to avoid having a statement prepared twice, avoid using PREPARE INTO when REOPT(ALWAYS) or REOPT(ONCE) is in effect.

Relationship of cursor attributes on PREPARE statements and SELECT or DECLARE CURSOR statements: Cursor attributes that are specified as part of the *select-statement* are used instead of any corresponding options that specified with the ATTRIBUTES clause on PREPARE. Attributes that are specified as part of the ATTRIBUTES clause of PREPARE take precedence over any corresponding option that is specified with the DECLARE CURSOR statement. The order for using cursor attributes is as follows:

- SELECT (highest priority)
- PREPARE statement ATTRIBUTES clause
- DECLARE CURSOR (lowest priority)

For example, assume that host variable MYQ has been set to the following SELECT statement:

```
SELECT WORKDEPT, EMPNO, SALARY, BONUS, COMM
     FROM EMP
    WHERE WORKDEPT IN ('D11', 'D21')
    FOR UPDATE OF SALARY, BONUS, COMM
```

If the following PREPARE statement were issued, then the FOR UPDATE clause specified as part of the SELECT statement would be used instead of the FOR READ ONLY clause specified with the ATTRIBUTES clause as part of the PREPARE statement. Thus, the cursor would be updatable.

```
attrstring = 'FOR READ ONLY';
EXEC SQL PREPARE stmt1 ATTRIBUTES :attrstring FROM :MYQ;
```

Effect of the CURRENT EXPLAIN MODE special register:

If the CURRENT EXPLAIN MODE special register is set to EXPLAIN, the statement is prepared for explain only and is not executable, unless the statement is a SET statement. Attempting to execute the prepared statement will return an error. See the "CURRENT EXPLAIN MODE" on page 175 special register for more information.

Precedence of attributes for SELECT and UPDATE WHERE CURRENT OF for positioned updates:

If an UPDATE WHERE CURRENT OF statement and the associated SELECT statement are both prepared and both statements have the same PREPARE attributes, the values of the PREPARE attributes for the UPDATE WHERE CURRENT OF statement override the values of the PREPARE attributes for the SELECT statement.

Effect of extended indicator PREPARE attributes on dynamically executed positioned updates:

If an UPDATE statement with the WHERE CURRENT OF clause and the associated SELECT statement are both prepared, if extended indicator variables are used depends on the WITH EXTENDED INDICATORS or WITHOUT EXTENDED INDICATORS attributes in each of the PREPARE statements.

Table 151. Interaction between EXTENDED INDICATOR attributes of PREPARE statements for SELECT and UPDATE statements

Extended indicator attribute of PREPARE for SELECT statement	Extended indicator attribute of PREPARE for UPDATE statement with WHERE CURRENT OF clause	Result
WITH EXTENDED INDICATORS	WITH EXTENDED INDICATORS	The PREPARE attributes of the UPDATE statement override the PREPARE attributes of the SELECT statement. Non-updatable columns can be in the select-list.
WITH EXTENDED INDICATORS	WITHOUT EXTENDED INDICATORS	The UPDATE statement is executed without extended indicator parameters.

Table 151. Interaction between EXTENDED INDICATOR attributes of PREPARE statements for SELECT and UPDATE statements (continued)

Extended indicator attribute of PREPARE for SELECT statement	Extended indicator attribute of PREPARE for UPDATE statement with WHERE CURRENT OF clause	Result
WITH EXTENDED INDICATORS	Default (without attribute specified)	The PREPARE attributes of the SELECT statement override the default PREPARE attributes for the UPDATE statement. Non-updatable columns can be in the select-list.
WITHOUT EXTENDED INDICATORS	WITH EXTENDED INDICATORS	The PREPARE attributes of the UPDATE statement override the PREPARE attributes of the SELECT statement. Non-updatable columns are not in the implicit or explicit select-list.
WITHOUT EXTENDED INDICATORS	WITHOUT EXTENDED INDICATORS	The UPDATE statement is executed without extended indicator parameters.
WITHOUT EXTENDED INDICATORS	Default (without attribute specified)	The PREPARE attributes of the SELECT statement override the default PREPARE attributes for the UPDATE statement. The UPDATE statement is executed without extended indicator parameters.
Default (without attribute specified)	WITH EXTENDED INDICATORS	The PREPARE attributes of the UPDATE statement override the PREPARE attributes of the SELECT statement. Non-updatable columns are not in the implicit or explicit select-list.
Default (without attribute specified)	WITHOUT EXTENDED INDICATORS	The PREPARE attributes of the UPDATE statement override the PREPARE attributes of the SELECT statement. The UPDATE statement is executed without extended indicator parameters.
Default (without attribute specified)	Default (without attribute specified)	The UPDATE statement is executed without extended indicator parameters.

Interactions between the SKIPUNCI subsystem parameter and the PREPARE statement: When the PREPARE statement is specified with either the CURRENTLY COMMITTED or WAIT FOR OUTCOME clauses and the subsystem parameter SKIPUNCI is in effect, the following table describes whether uncommitted inserts are skipped, or if the transaction will wait until a commit or rollback before completing:

Table 152. Interaction between SKIPUNCI subsystem parameter and PREPARE statement

Value of SKIPUNCI subsystem parameter	PREPARE statement attributeworking	Skip uncommitted inserts, or wait for commit or rollback
YES	CURRENTLY COMMITTED	Skip
YES	WAIT FOR OUTCOME	Wait
YES	Not specified	Skip
NO	CURRENTLY COMMITTED	Skip
NO	WAIT FOR OUTCOME	Wait
NO	Not specified	Wait

Extended indicator variables and deferred error checks:

When extended indicator variables are enabled, the indicator value of unassigned causes the associated target column to be omitted from the statement. Because of that, validation that is normally done in statement preparation (to recognize an INSERT into, or UPDATE of, a non-updatable column) is deferred until statement execution. If statement validation fails, an error is returned when the statement is run, not when the statement is prepared.

Reuse of prepared statements in the dynamic statement cache with CONCENTRATE STATEMENTS WITH LITERALS

: To be eligible for reuse of constants, the constants in both the new statement and the cached statement must have the same:

1. immediate usage context
2. data type
3. data type length and size

If DB2 determines that both instances of the constant meet the criteria for reuse, a cached statement that is prepared using the **CONCENTRATE STATEMENTS WITH LITERALS** option can be shared by the same SQL statement with different constants. Even though the new dynamic SQL statement will share the cached statement, the new statement will use its own literal constants when the statement is run, not the constants of the cached statement.

There are some exceptions. For example, the built-in function SUBSTR, for which, because of the immediate usage context, constant reuse in the cached statement that uses a different constant value can not be done without the risk of returning incorrect output or results. In such cases, only an SQL statement instance with the exact same constant value as the cached version of the statement is eligible for reuse. DB2 determines when and where this immediate usage context restriction applies.

When the **CONCENTRATE STATEMENTS WITH LITERALS** option is specified, DB2 considers the values of the literal constants for access path selection only for statements that are bound with the REOPT(ONCE) or REOPT(AUTO) bind options.

The DECFLOAT defined constants NAN, SNAN, and INFINITY can qualify for literal constant reuse.

The following examples show how PREPARE is used with **CONCENTRATE STATEMENTS WITH LITERALS**. X, Y, and Z are columns of defined as DECIMAL data type:

```
DECLARE C1 CURSOR
 FOR DYNSQL_WITH_LITERAL;

DYNSQL_SELECT = 'SELECT X, Y, Z
        FROM TABLE1
        WHERE X < 9';

attrstring = 'CONCENTRATE STATEMENTS WITH LITERALS';

EXEC SQL PREPARE DYNSQL_WITH_LITERAL
 ATTRIBUTES :attrstring
 FROM :DYNSQL_SELECT;

EXEC SQL OPEN C1;

DYNSQL_INSERT = 'INSERT INTO
        TABLE1 (X, Y, Z)
        VALUES (8,109,29)';

attrstring = 'CONCENTRATE STATEMENTS WITH LITERALS';

EXEC SQL PREPARE DYNSQL_INSERT_WITH_LITERAL
  ATTRIBUTES :attrstring
  FROM :DYNSQL_INSERT;

EXEC SQL EXECUTE DYNSQL_INSERT_WITH_LITERAL;
```

Examples

Example 1: In this PL/I example, an INSERT statement with parameter markers is prepared and executed. Before execution, values for the parameter markers are read into the host variables S1, S2, S3, S4, and S5.

```
EXEC SQL PREPARE DEPT_INSERT FROM
  'INSERT INTO DSN8B10.DEPT VALUES(?,?,?,?,?)';
-- Check for successful execution and read values into host variables
EXEC SQL EXECUTE DEPT_INSERT USING :S1, :S2, :S3, :S4, :S5;
```

Example 2: Prepare a dynamic SELECT statement specifying the attributes of the cursor with a host variable on the PREPARE statement. Assume that the text of the SELECT statement is in a variable named stmttxt, and that the attributes of the cursor are in a variable named attrvar.

```
EXEC SQL DECLARE mycursor CURSOR FOR mystmt;
EXEC SQL PREPARE mystmt ATTRIBUTES :attrvar
            FROM :stmttxt;
EXEC SQL DESCRIBE mystmt INTO :mysqlda;
EXEC SQL OPEN mycursor;
EXEC SQL FETCH FROM mycursor USING DESCRIPTOR :mysqlda;
```

REFRESH TABLE

The REFRESH TABLE statement refreshes the data in a materialized query table. The statement deletes all rows in the materialized query table, executes the fullselect in the table definition to recalculate the data from the tables specified in the fullselect, inserts the calculated result into the materialized query table, and updates the catalog for the refresh timestamp and cardinality of the table. The table can exist at the current server or at any DB2 subsystem with which the current server can establish a connection.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

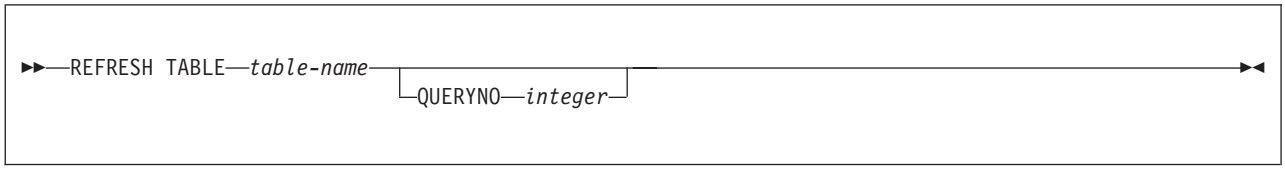
The privilege set for REFRESH TABLE must include at least one of the following authorities:

- Ownership of the materialized query table
- DBADM or DBCTRL authority on the database that contains the materialized query table
- SYSADM or SYSCTRL authority
- DATAACCESS authority

If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the statements dynamically prepared, the privilege set is determined by the DYNAMICRULES behavior in effect (run, bind, define, or invoke). For more information on these behaviors, including a list of the DYNAMICRULES bind option values, see "Authorization IDs and dynamic SQL" on page 75.

Syntax



Description

table-name
Identifies the table to be refreshed. The name must identify a materialized query table. REFRESH TABLE evaluates the fullselect in the *materialized-query-definition* clause to refresh the table. The isolation level for the fullselect is the isolation level of the materialized query table recorded when CREATE TABLE or ALTER TABLE was issued.

QUERYNO *integer*
Specifies the number to be used for this SQL statement in EXPLAIN output

and trace records. The number is used for the QUERYNO column of the plan table for the rows that contain information about this SQL statement. This number is also used in the QUERYNO column of the SYSIBM.SYSSTMT and SYSIBM.SYSPACKSTMT catalog tables.

If the clause is omitted, the number associated with the SQL statement is the statement number assigned during precompilation. Thus, if the application program is changed and then precompiled, that statement number might change.

Notes

Automatic query rewrite using materialized query tables is not attempted for the fullselect in the materialized query table definition during the processing of REFRESH TABLE statement.

After successful execution of a REFRESH TABLE statement, the SQLCA field SQLERRD(3) will contain the number of rows inserted into the materialized query table.

The EXPLAIN output for REFRESH TABLE *table-name* is the same as the EXPLAIN output for INSERT INTO *table-name fullselect* where *fullselect* is from the materialized query table definition.

If the materialized query table has a security label column, the REFRESH TABLE statement does not do any checking for multilevel security with row-level granularity when it deletes and repopulates the data in the table by executing the fullselect. Instead, DB2 performs the checking for multilevel security with row-level granularity when the materialized query table is exploited in automatic query rewrite or is used directly.

The REFRESH TABLE statement can be used to remove a table space from the logical page list and reset recover-pending status. This can only be done by using REFRESH TABLE to repopulate a materialized query table where the materialized query table is the only table in the table space.

Example

Issue a statement to refresh the content of a materialized query table that is named SALESCOUNT. The statement recalculates the data from the fullselect that was used to define SALESCOUNT and refreshes the content of SALESCOUNT with the recalculated results.

```
REFRESH TABLE SALESCOUNT;
```

RELEASE (connection)

The RELEASE (connection) statement places one or more connections in the release pending state.

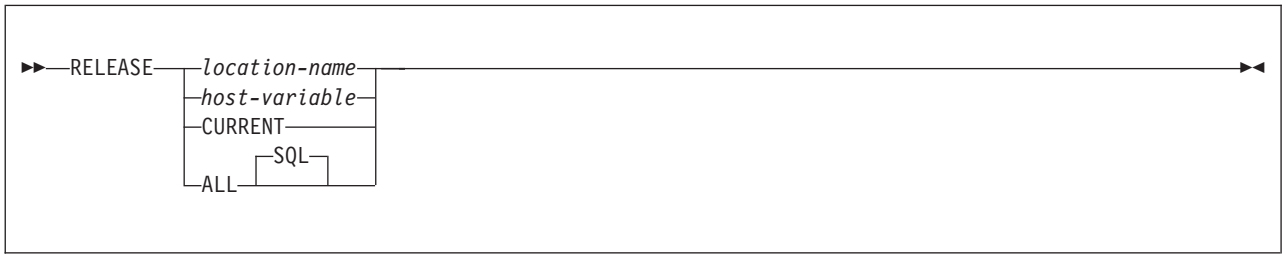
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

None required.

Syntax



Description

location-name **or** *host-variable*
Identifies an SQL connection by the specified location name or the location name contained in the host variable. If a host variable is specified:

- It must be a character string variable with a length attribute that is not greater than 16. (A C NUL-terminated character string can be up to 17 bytes.)
- It must not be followed by an indicator variable.
- The location name must be left-justified within the host variable and must conform to the rules for forming an ordinary location identifier.
- If the length of the location name is less than the length of the host variable, it must be padded on the right with blanks.

The specified location name or the location name contained in the host variable must identify an existing SQL connection of the application process.

CURRENT
Identifies the current SQL connection of the application process. The application process must be in the connected state.

ALL or ALL SQL
Identifies all existing connections (including local, and SQL) of the application process. An error or warning does not occur if no connections exist when the statement is executed.

If the RELEASE (connection) statement is successful, each identified connection is placed in the release-pending state and, therefore, will be ended during the next commit operation. If the RELEASE (connection) statement is unsuccessful, the

connection state of the application process and the states of its connections are unchanged.

Notes

RELEASE and CONNECT (Type 1): Using CONNECT (Type 1) semantics does not prevent using RELEASE (connection).

Scope of RELEASE: RELEASE (connection) does not close cursors, does not release any resources, and does not prevent further use of the connection.

Resource considerations for remote connections: Resources are required to create and maintain remote connections. Thus, a remote connection that is not going to be reused should be in the release pending status and one that is going to be reused should not be in the release pending status. Remote connections can also be ended during a commit operation as a result of the DISCONNECT(AUTOMATIC) or DISCONNECT(CONDITIONAL) bind option.

If the current SQL connection is in the release pending status when a commit operation is performed, the connection is ended and the application process is in the unconnected state. In this case, the next executed SQL statement should be CONNECT or SET CONNECTION.

Connection states: ROLLBACK does not reset the state of a connection from release pending to held.

If the current SQL connection is in the release pending state when a commit operation is performed, the connection is ended and the application process is in the unconnected state. In this case, the next executed SQL statement must be CONNECT or SET CONNECTION.

For further information, see “Application process connection states” on page 39.

Location names CURRENT and ALL: A database server named CURRENT or ALL can only be identified by a host variable or a delimited identifier. A connection in the release pending state is ended during a commit operation even though it has an open cursor defined with WITH HOLD.

Encoding scheme of a host variable: If the RELEASE statement contains host variables, the contents of the host variables are assumed to be in the encoding scheme that was specified in the ENCODING parameter when the package or plan that contains the statement was bound.

Examples

Example 1: The SQL connection to TOROLAB1 is not needed in the next unit of work. The following statement causes it to be ended during the next commit operation:

```
EXEC SQL RELEASE TOROLAB1;
```

Example 2: The current SQL connection is not needed in the next unit of work. The following statement causes it to be ended during the next commit operation:

```
EXEC SQL RELEASE CURRENT;
```

RELEASE SAVEPOINT

The RELEASE SAVEPOINT statement releases the identified savepoint and any subsequently established savepoints within a unit of recovery.

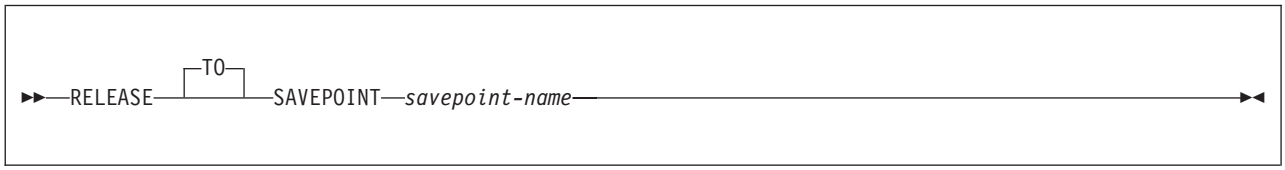
Invocation

This statement can be imbedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

savepoint-name
Identifies the savepoint to release. If the named savepoint does not exist, an error occurs. The name must identify a savepoint that exists at the current server. After a savepoint is released, it is no longer maintained and rollback to the savepoint is no longer possible.

Notes

Savepoint names: The name of the savepoint that was released can be reused in another SAVEPOINT statement, regardless of whether the UNIQUE keyword was specified on an earlier SAVEPOINT statement that specified this same savepoint name.

Example

Assume that a main routine sets savepoint A and then invokes a subroutine that sets savepoints B and C. When control returns to the main routine, release savepoint A and any subsequently set savepoints. Savepoints B and C, which were set by the subroutine, are released in addition to A.

```
⋮
  RELEASE SAVEPOINT A;
```

RENAME

The RENAME statement renames an existing table or index.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

To rename a table, the privilege set that is defined below must include at least one of the following privileges:

- Ownership of the table
- DBADM, DBCTRL, or DBMAINT authority for the database that contains the table
- SYSADM or SYSCTRL authority
- System DBADM

If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

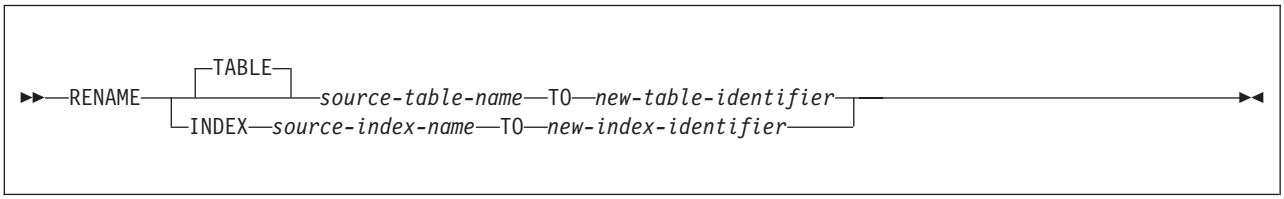
To rename an index, the privilege set that is defined below must include at least one of the following privileges:

- Ownership of the table for which the index is defined
- Ownership of the index that is being renamed
- DBADM, DBCTRL, or DBMAINT authority for the database that contains the index
- SYSADM or SYSCTRL authority
- System DBADM

If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set is the union of the privilege sets that are held by each authorization ID of the process.

Syntax



Description

source-table-name
Identifies the existing table that is to be renamed. The name, including the

implicit or explicit qualifier, must identify a table that exists at the current server. The name must not identify any of the following types of tables:

- A declared temporary table
- A catalog table
- An active resource limit specification table
- A materialized query table
- A clone table
- A system-period temporal table
- A history table for a system-period temporal table
- A table with a trigger defined on it
- A table that is referenced in the definition of a row permission
- A table that is referenced in the definition of a column mask
- A view
- A synonym
- An archive-enabled table
- An archive table

If you specify a three-part name or alias for the source table, the source table must exist at the current server. If any view definitions or materialized query table definitions currently reference the source table, an error occurs.

new-table-identifier
Specifies the new name for the table without a qualifier. The qualifier of the *source-table-name* is used to qualify the new name for the table. The qualified name must not identify a table, view, alias, or synonym that exists at the current server, or a table that exists in the SYSIBM.SYSPENDINGOBJECTS catalog table.

source-index-name
Identifies the existing index that is to be renamed. The name, including an implicit or explicit qualifier, must identify an index that exists at the current server. The name must not identify a system defined catalog index, an index for a declared temporary table, or an index for an active resource limit specification table.

new-index-identifier
Specifies that new name for the index without a qualifier. The qualifier of the *source-index-name* is used to qualify the new name for the index. The qualified name must not identify an index that exists at the current server or an index that exists in the SYSIBM.SYSPENDINGOBJECTS catalog table.

Notes

Effects of the statement: The specified table or index is renamed to the new name. For a renamed table, all privileges and indexes on the table are preserved. For a renamed index, all privileges are preserved.

Invalidation of packages: When any table except an auxiliary table is renamed, packages that refer to that table are invalidated. When an auxiliary table is renamed, packages that refer to the auxiliary table are not invalidated.

Restriction when there are pending changes to the definition: A RENAME INDEX statement is not allowed if there are pending changes to the definition of the index.

Considerations for aliases: If an alias name is specified for *table-name*, the table must exist at the current server, and the table that is identified by the alias is renamed. The name of the alias is not changed and continues to refer to the old table name after the rename.

Changing the name of an alias with the RENAME statement is not supported. To change the name to which an alias refers, you must drop the alias and then re-create it.

Considerations for plan tables: The RENAME INDEX statement does not update the contents of a plan table. Rows that exist in a plan table that are generated from a EXPLAIN statement can contain the name of an index in the access path selections. When an index is renamed, any entries in existing plan tables that refer to the old index name are not updated.

Transfer of authorization, referential integrity constraints, and indexes: *All* authorizations associated with the source table name are *transferred* to the new (target) table name. The authorization catalog tables are updated appropriately.

Referential integrity constraints involving the source table are updated to refer to the new table. The catalog tables are updated appropriately.

Indexes that are defined for the source table are *transferred* to the new table. The index catalog tables are updated appropriately.

Object identifier: Renamed tables and indexes keep the same object identifier as the original table or index.

Renaming registration tables: If an application registration table (ART) or object registration table (ORT) or an index of an ART or ORT is specified as the source table for RENAME, when RENAME completes, it is as if that table had been dropped. There is no ART or ORT once the ART or ORT table has been renamed.

Catalog table updates: Entries in the following catalog tables are updated to reflect the new table:

- SYSAUXRELS
- SYSCHECKS
- SYSCHECKS2
- SYSCHECKDEP
- SYSCOLAUTH
- SYSCOLDIST
- SYSCOLDIST_HIST
- SYSCOLDISTSTATS
- SYSCOLSTATS
- SYSCOLUMNS
- SYSCOLUMNS_HIST
- SYSCONSTDEP
- SYSFIELDS
- SYSFOREIGNKEYS
- SYSINDEXES
- SYSINDEXES_HIST
- SYSKEYCOLUSE

- SYSPLANDEP
- SYSPACKDEP
- SYSRELS
- SYSSEQUENCESDEP
- SYSSYNONYMS
- SYSTABAUTH
- SYSTABCONST
- SYSTABLES
- SYSTABLES_HIST
- SYSTABSTATS
- SYSTABSTATS_HIST

Entries in SYSSTMT and SYSPACKSTMT are not updated.

Entries in the following catalog tables are updated to reflect the new index:
- SYSDEPENDENCIES
- SYSINDEXES
- SYSINDEXES_HIST
- SYSINDEXESPART
- SYSINDEXESPART_HIST
- SYSINDEXSPACESTATS
- SYSINDEXSTATS
- SYSINDEXSTATS_HIST
- SYSKEYS
- SYSKEYTARGETS
- SYSKEYTARGETS_HIST
- SYSKEYTARGETSTATS
- SYSKEYTGTDIST
- SYSKEYTGTDIST_HIST
- SYSKEYTGTDISTSTATS
- SYSOBJROLEDEP
- SYSPACKDEP
- SYSPLANDEP
- SYSRELS
- SYSTABCONST
- SYSTABLEPART

Examples

Example 1: Change the name of the EMP table to EMPLOYEE:

```
RENAME TABLE EMP TO EMPLOYEE;
```

Example 2: Change the name of the EMP_USA_HIS2002:

```
RENAME TABLE EMP_USA_HIS2002 TO EMPLOYEE_UNITEDSTATES_HISTORY2002;
```

Example 3: Change the name of the EMPINDX1 to EMPLOYEE_INDEX:

```
RENAME INDEX COMPANY.EMPINDX1 TO EMPLOYEE_INDEX;
```

REVOKE

The REVOKE statement revokes privileges from authorization IDs. There is a separate form of the statement for each of these classes of privilege:

- Collection
- Database
- Distinct type
- Function or stored procedure
- Package
- Plan
- Schema
- Sequence
- System
- Table or view
- Use

The applicable objects are always at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

If the authorization mechanism was not activated when the DB2 subsystem was installed, an error condition occurs.

Authorization

If the BY clause is not specified, the authorization ID of the statement must have granted at least one of the specified privileges to every *authorization-name* specified in the FROM clause (including PUBLIC, if specified). If the BY clause is specified, the authorization ID of the statement must have SECADM or ACCESSCTRL authority.

Note: If installation parameter SEPARATE SECURITY is NO, SYSADM authority has implicit SECADM authority and SYSCTRL authority has implicit ACCESSCTRL authority.

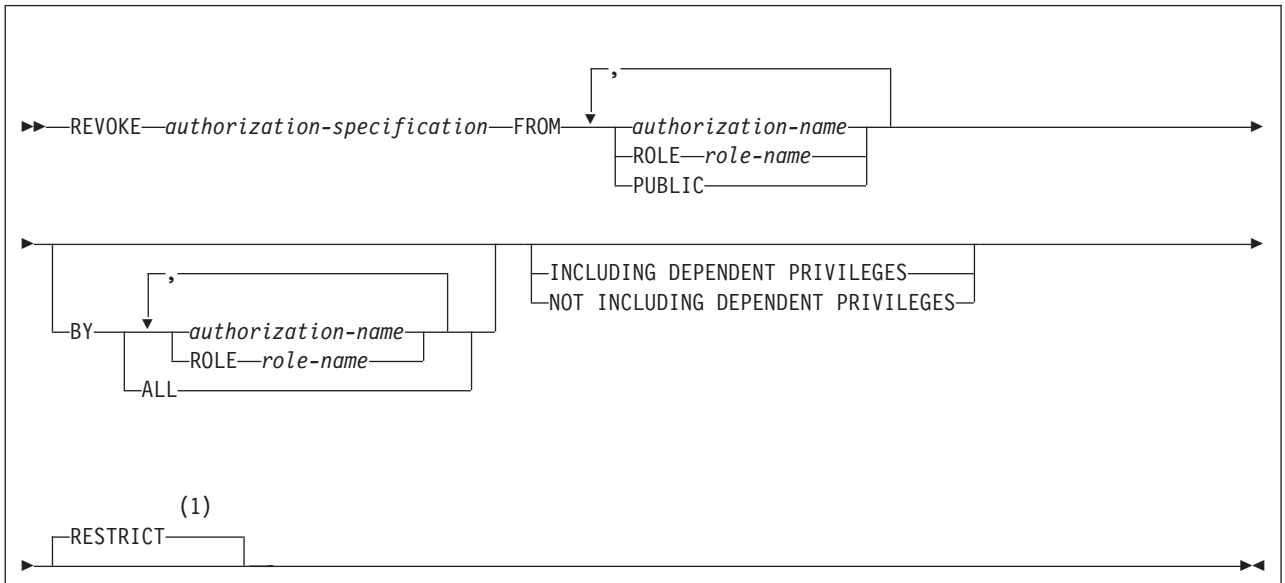
If the BY clause is specified and the privilege set includes ACCESSCTRL, all privileges and authorities can be revoked except for the following:

- System DBADM
- ACCESSCTRL
- DATAACCESS
- CREATE_SECURE_OBJECT privilege

If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. The owner can be a role. If the statement is dynamically prepared, the privilege set is the privileges that are held by the SQL authorization ID of the process. However, if the process is running in a trusted context that is defined with the ROLE AS OBJECT OWNER

AND QUALIFIER CLAUSE, the privilege set is the privileges that are held by the role that is in effect.

Syntax



Notes:

1 The RESTRICT clause is the default only for the forms of the REVOKE statement that allow it.

Description

authorization-specification
Specifies one or more privileges for the class of privilege. The same privilege must not be specified more than once.

FROM
Specifies from what authorization IDs the privileges are revoked.

authorization-name,...
Lists one or more authorization IDs. Do not use the same authorization ID more than one time. If the *authorization-name* is specified in lowercase, it must be delimited using double quotes.

The value of CURRENT RULES determines if you can use the ID of the REVOKE statement itself (to revoke privileges from yourself). When CURRENT RULES is:

DB2 You cannot use the ID of the REVOKE statement.

STD
You can use the ID of the REVOKE statement.

ROLE *role-name*
Lists one or more roles. Do not specify the same role more than one time.

PUBLIC
Revokes a grant of privileges to PUBLIC.

BY Lists grantors who have granted privileges and revokes each named privilege that was explicitly granted to some named user by one of the named grantors.

Only an authorization ID or role with SYSADM or SYSCTRL authority can use BY, even if the authorization ID or role names only itself in the BY clause.

authorization-name,...
Lists one or more authorization IDs of users who were the grantors of the privileges named. Do not use the same authorization ID more than once. Each grantor that is listed must have explicitly granted some named privilege to all of the named users or roles.

ROLE *role-name*
Lists one or more roles that were the grantors of the privileges named. Do not specify the same role more than one time. Each grantor that is listed must have explicitly granted some named privilege to all of the named users or roles.

ALL
Revokes each named privilege from all named users who were explicitly granted the privilege, regardless of who granted it.

INCLUDING DEPENDENT PRIVILEGES or NOT INCLUDING DEPENDENT PRIVILEGES
Specifies whether revoking a privilege or an authority from an authorization ID or a role also results in revoking the grants that were made by that user. The default value is based on the authority that is being revoked and the REVOKE_DEP_PRIVILEGES system parameter:

- When ACCESSCTRL, DATAACCESS, or system DBADM authority is revoked, **NOT INCLUDING DEPENDENT PRIVILEGES** is assumed and the clause must be specified on the REVOKE statement.
- When the REVOKE_DEP_PRIVILEGES system parameter is set to NO, **NOT INCLUDING DEPENDENT PRIVILEGES** is assumed and an error is returned if the statement includes **INCLUDING DEPENDENT PRIVILEGES**.
- Otherwise, **INCLUDING DEPENDENT PRIVILEGES** is assumed and the clause must be specified on the REVOKE statement.

INCLUDING DEPENDENT PRIVILEGES
Specifies that revoking a privilege or an authority from an authorization ID or a role also results in revoking dependent privileges. This means that any grants that were made by the user will continue to be revoked, until all grants in the chain have been revoked.

INCLUDING DEPENDENT PRIVILEGES cannot be specified if the system parameter REVOKE_DEP_PRIVILEGES is set to NO, which enforces the behavior to not include the dependent privileges.

NOT INCLUDING DEPENDENT PRIVILEGES
Specifies that revoking a privilege or an authority from an authorization ID or a role does not cause the grants that were made by the user to be revoked. However, for the revoked privileges, all implications of the privilege being revoked are applied. For example, if the revoked privileges were required to bind a package successfully, that package would continue to be invalidated as a result of the package owner losing these privileges. An object might be dropped if a privilege is revoked that was used to create the object.

NOT INCLUDING DEPENDENT PRIVILEGES must be specified when ACCESSCTRL, DATAACCESS, or system DBADM authority is revoked.

NOT INCLUDING DEPENDENT PRIVILEGES cannot be specified if the system parameter REVOKE_DEP_PRIVILEGES is set toYES, which enforces the behavior to include dependent privileges in the revoke.

RESTRICT
Prevents the named privilege from being revoked when certain conditions apply. RESTRICT is the default only for the forms of the REVOKE statement that allow it. These forms are revoking the USAGE privilege on distinct types, the EXECUTE privilege on user-defined functions and stored procedures, and the USAGE privilege on sequences.

Notes

Revoked privileges: The privileges revoked from an authorization ID or a role are those that are identified in the statement and which were granted to the user by the grantor. Other privileges can be revoked as the result of revoking dependent privileges. For more on DB2 privileges, see *DB2 Administration Guide*.

Revoke dependent privileges: Revoking a privilege from a user can also cause that privilege to be revoked from other users. This was previously known as *cascade revoke*. When revoking a privilege from an authorization ID or a role, DB2 looks for and revokes any grants of the privilege where the grantor is the same as the authorization ID or role of the original revoke. The following rules must be true for privilege P' to be revoked from U3 when U1 revokes privilege P from U2:

- P and P' are the same privilege.
- U2 granted privilege P' to U3.
- No one granted privilege P to U2 prior to the grant by U1.
- U2 does not have installation SYSADM authority.

The rules also apply to the implicit grants that are made as a result of a CREATE VIEW statement.

Revoking dependent privileges does not occur under any of the following conditions:

- The privilege was granted by a current installation SYSADM user.
- The privilege is the USAGE privilege on a distinct type and the revokee owns any of these items:
  - A user-defined function or stored procedure that uses the distinct type
  - A table that has a column that uses the distinct type
  - A sequence whose data type is the distinct type
- The privilege is the USAGE privilege on a sequence and the revokee owns any of these items:
  - A trigger that has a NEXT VALUE or PREVIOUS VALUE expression that specifies the sequence
  - An inline SQL function that has a NEXT VALUE or PREVIOUS VALUE expression in the function body that specifies the sequence
- The privilege is the EXECUTE privilege on a user-defined function and the revokee owns any of these items:
  - A user-defined function that is sourced on the function
  - A view that uses the function
  - A trigger package that uses the function
  - A table that uses the function in a check constraint or a user-defined default type
- The privilege is the EXECUTE privilege on a stored procedure and the revokee owns any of these items:

– A trigger package that refers to the stored procedure in a CALL statement.

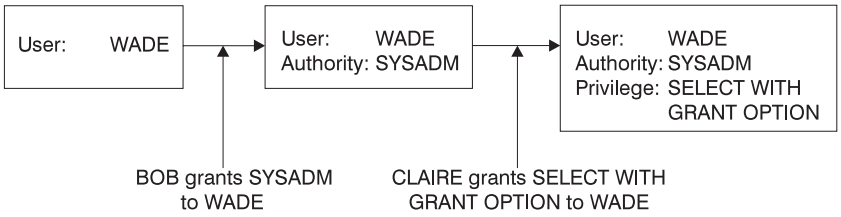
- If the ACCESSCTRL administrative authority is revoked from a user, grants that are made by this ACCESSCTRL user are not revoked.

  If this user revoked grants made by it, those revokes will continue to revoke the dependent privileges, unless the behavior to not include the dependent privileges was specified either by using the system parameter REVOKE_DEP_PRIVILEGES or by using the REVOKE statement if REVOKE_DEP_PRIVILEGES is set to SQLSTMT.
- If SECADM is removed from a user, grants that are made by this SECADM user are not revoked.

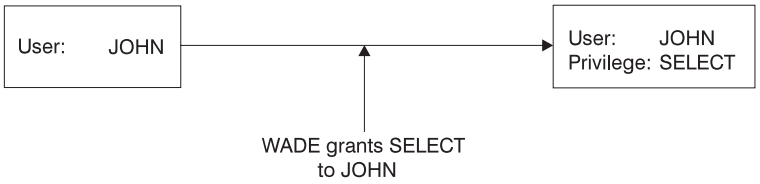
  If this user revoked grants made by it, those revokes will continue to revoke the dependent privileges, unless the behavior to not include the dependent privileges was specified either by using the system parameter REVOKE_DEP_PRIVILEGES or by using the REVOKE statement if REVOKE_DEP_PRIVILEGES is set to SQLSTMT.
- If **NOT INCLUDING DEPENDENT PRIVILEGES** option is specified, the grants made by this user are not revoked.

Refer to the diagrams for the following example:

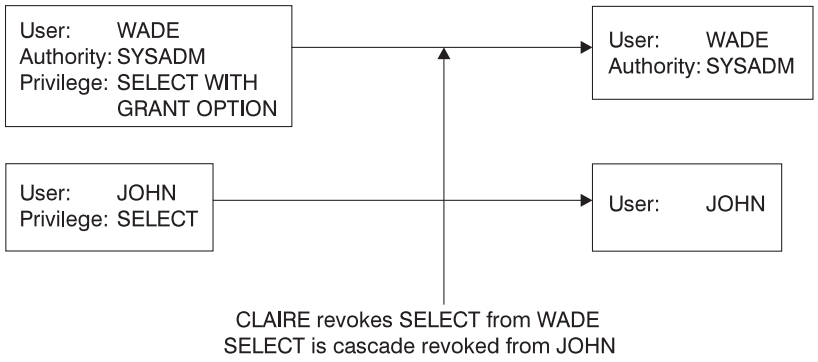
1. Suppose BOB grants SYSADM authority to WADE. Later, CLAIRE grants the SELECT privilege on a table with the WITH GRANT OPTION to WADE.



2. WADE grants the SELECT privilege to JOHN on the same table.



3. When CLAIRE revokes the SELECT privilege on the table from WADE, the SELECT privilege on that table is also revoked from JOHN.



The grant from WADE to JOHN is removed because WADE had not been granted the SELECT privilege from any other source before CLAIRE made the grant. The SYSADM authority granted to WADE from BOB does not affect the cascade revoke. For more on SYSADM and installation SYSADM authority, see *DB2 Administration Guide*. For another example of cascading revokes, see *DB2 Administration Guide*.

Revoking a SELECT privilege that was exercised to create a view or materialized query table causes the view to be dropped, unless the owner of the view was directly granted the SELECT privilege from another source before the view was created. Revoking a SYSADM privilege that was required to create a view causes the view to be dropped. For details on when SYSADM authority is required to create a view, see *Authorization* in "CREATE VIEW" on page 1527.

Invalidation of packages: A revoke or cascaded revoke of any privilege or role that was exercised to create a package makes the package invalid when the revokee no longer holds the privilege from any other source. Corresponding authorization caches are cleared even if the revokee has the privilege from any other source. [37]

Inoperative packages: A revoke or cascaded revoke of the EXECUTE privilege on a user-defined function that was exercised to create a package makes the package inoperative and causes the corresponding authorization caches to be cleared when the revokee no longer holds the privilege from any other source.[37]

Privileges belonging to an authority: You can revoke an administrative authority, but you cannot separately revoke the specific privileges inherent in that administrative authority.

Let P be a privilege inherent in authority X. A user with authority X can also have privilege P as a result of an explicit grant of P. In this case:

- If X is revoked, the user still has privilege P.
- If P is revoked, the user still has the privilege because it is inherent in X.

Revoking of privileges in a trusted context: Revokes that are made in a trusted context that is defined with the ROLE AS OBJECT OWNER clause result in the revoker being the role in effect. If the statement is prepared dynamically, the revoker is the role that is associated with the user that is running the statement. If the statement is embedded in a program, the revoker is the owner of the plan or

37. Dependencies on stored procedures can be checked only if the procedure name is specified as a constant and not via a host variable in the CALL statement.

package. If the ROLE AS OBJECT OWNER clause is not specified for the trusted context, the revoker is the authorization ID of the process.

Ownership privileges: The privileges inherent in the ownership of an object cannot be revoked.

Revoke not including dependent privileges: When a privilege is revoked from a user by specifying **NOT INCLUDING DEPENDENT PRIVILEGES**, the grants that were made by this user are not revoked and the grantor remains unchanged. If that user is later granted the same privilege and then this privilege is revoked by specifying **INCLUDING DEPENDENT PRIVILEGES**, that would also revoke all the grants that were previously made by this user. Refer to the following examples:

User U1 is granted SELECT on table T1 with GRANT OPTION:

1. U1 grants this privilege to U2.
2. SELECT privilege is revoked from U1 without including dependent privileges. As a result, the grant from U1 to U2 is not revoked.
3. U1 is again granted SELECT on T1.
4. SELECT is now revoked from U1 with including dependent privileges and the grant from U1 to U2 is now revoked.

1.

User U1 is granted SYSADM authority:

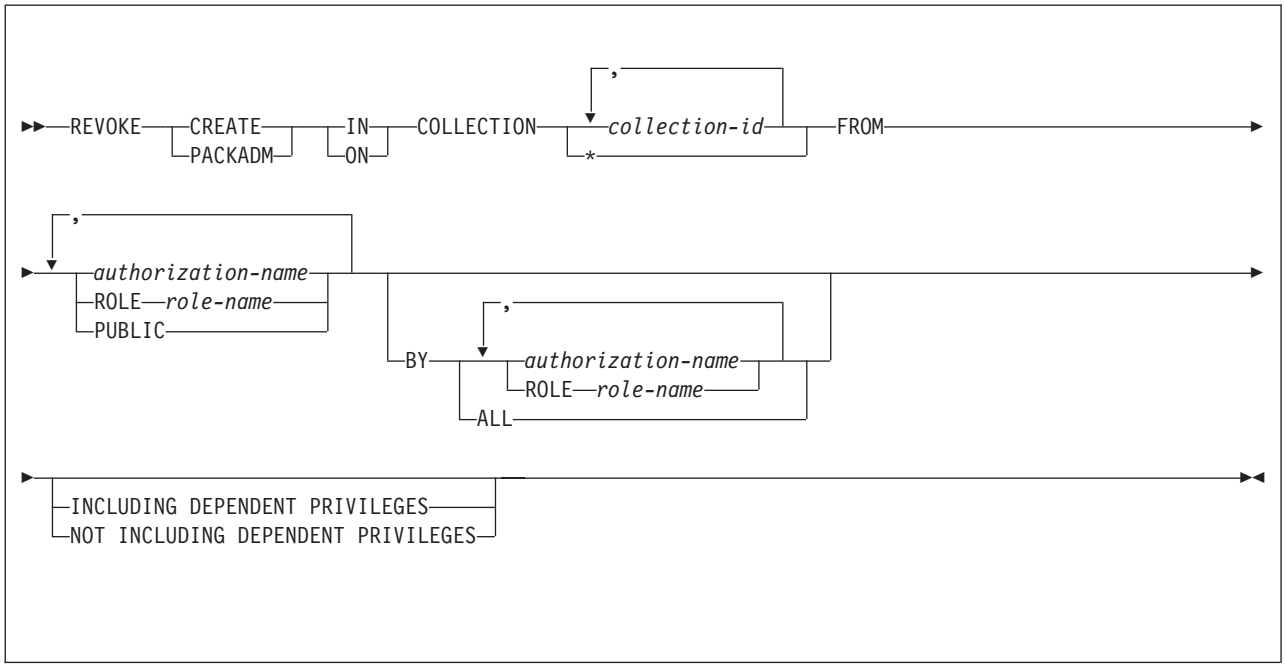
1. U1 grants privilege P1 to U2 and privilege P2 to U3.
2. SYSADM is revoked from U1 without including dependent privileges. The grants of privileges P1 and P2 to U2 and U3 are not revoked.
3. U1 is again granted SYSADM. U1 grants privilege P3 to U3.
4. SYSADM is now revoked from U1 including dependent privileges. Now, P1 granted to U2 and P2 and P3 granted to U3 are also revoked.

1.

REVOKE (collection privileges)

This form of the REVOKE statement revokes privileges on collections.

Syntax



Description

CREATE IN
Revokes the privilege to use the BIND subcommand to create packages in the designated collections.

The word ON can be used instead of IN.

PACKADM ON
Revokes the package administrator authority for the designated collections.

The word IN can be used instead of ON.

COLLECTION *collection-id,...*
Identifies the collections on which the specified privilege is revoked. For each identified collection, you (or the indicated grantors) must have granted the specified privilege on that collection to all identified users (including PUBLIC if specified). The same collection must not be identified more than once.

COLLECTION *
Indicates that the specified privilege on COLLECTION * is revoked. You (or the indicated grantors) must have granted the specified privilege on COLLECTION * to all identified users (including PUBLIC if specified). Privileges granted on specific collections are not affected.

FROM
Refer to "REVOKE" on page 1812 for a description of the FROM clause.

BY Refer to "REVOKE" on page 1812 for a description of the BY clause.

INCLUDING DEPENDENT PRIVILEGES or NOT INCLUDING DEPENDENT PRIVILEGES
Specifies whether revoking a privilege or an authority from an authorization

ID or a role also results in revoking the grants that were made by that user. The default value is based on the authority that is being revoked and the REVOKE_DEP_PRIVILEGES system parameter:

- When ACCESSCTRL, DATAACCESS, or system DBADM authority is revoked, **NOT INCLUDING DEPENDENT PRIVILEGES** is assumed and the clause must be specified on the REVOKE statement.
- When the REVOKE_DEP_PRIVILEGES system parameter is set to NO, **NOT INCLUDING DEPENDENT PRIVILEGES** is assumed and an error is returned if the statement includes **INCLUDING DEPENDENT PRIVILEGES**.
- Otherwise, **INCLUDING DEPENDENT PRIVILEGES** is assumed and the clause must be specified on the REVOKE statement.

INCLUDING DEPENDENT PRIVILEGES
Specifies that revoking a privilege or an authority from an authorization ID or a role also results in revoking dependent privileges. This means that any grants that were made by the user will continue to be revoked, until all grants in the chain have been revoked.

INCLUDING DEPENDENT PRIVILEGES cannot be specified if the system parameter REVOKE_DEP_PRIVILEGES is set to NO, which enforces the behavior to not include the dependent privileges.

NOT INCLUDING DEPENDENT PRIVILEGES
Specifies that revoking a privilege or an authority from an authorization ID or a role does not cause the grants that were made by the user to be revoked. However, for the revoked privileges, all implications of the privilege being revoked are applied. For example, if the revoked privileges were required to bind a package successfully, that package would continue to be invalidated as a result of the package owner losing these privileges. An object might be dropped if a privilege is revoked that was used to create the object.

NOT INCLUDING DEPENDENT PRIVILEGES must be specified when ACCESSCTRL, DATAACCESS, or system DBADM authority is revoked.

NOT INCLUDING DEPENDENT PRIVILEGES cannot be specified if the system parameter REVOKE_DEP_PRIVILEGES is set toYES, which enforces the behavior to include dependent privileges in the revoke.

Examples

Example 1: Revoke the privilege to create new packages in collections QAACLONE and DSN8CC61 from CLARK.

```
REVOKE CREATE IN COLLECTION QAACLONE, DSN8CC61 FROM CLARK;
```

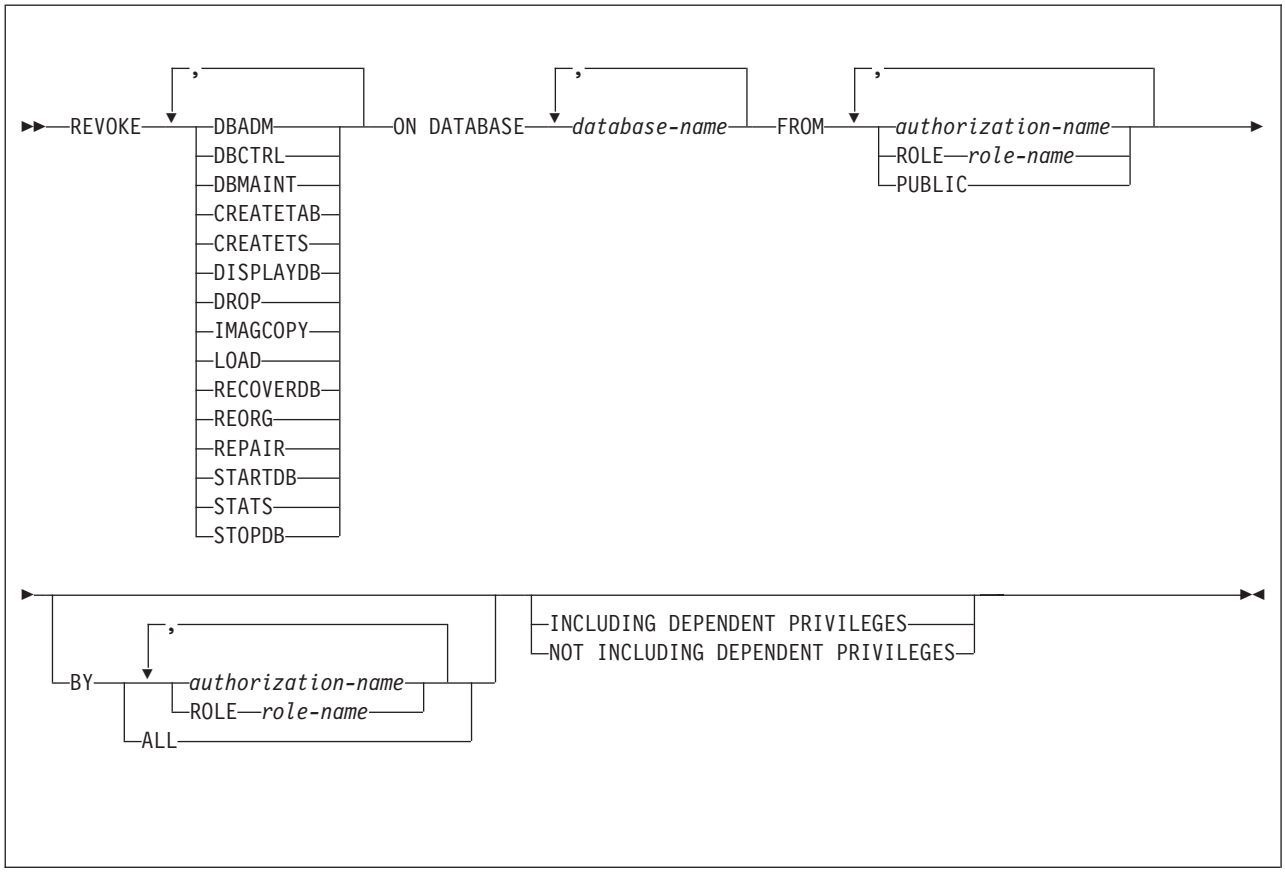
Example 2: Revoke the privilege to create new packages in collections DSN8CC91 from role ROLE1:

```
REVOKE CREATE IN COLLECTION DSN8CC91 FROM ROLE ROLE1;
```

REVOKE (database privileges)

This form of the REVOKE statement revokes database privileges.

Syntax



Description

Each keyword listed revokes the privilege described, but only as it applies to or within the databases named in the statement.

DBADM
Revokes the database administrator authority.

DBCTRL
Revokes the database control authority.

DBMAINT
Revokes the database maintenance authority.

CREATETAB
Revokes the privilege to create new tables. If CREATETAB privilege is revoked from DSNDB04, tables cannot be created in implicitly created databases. For a work file database, you cannot revoke the privilege from PUBLIC. When a work file database is created, PUBLIC implicitly receives the CREATETAB privilege (without GRANT authority); this privilege is not recorded in the DB2 catalog, and it cannot be revoked.

CREATETS
Revokes the privilege to create new table spaces.

DISPLAYDB
Revokes the privilege to issue the DISPLAY DATABASE command.

DROP
Revokes the privilege to issue the DROP or ALTER statements in the specified databases.

IMAGCOPY
Revokes the privilege to run the COPY, MERGECOPY, and QUIESCE utilities against table spaces of the specified databases, and to run the MODIFY RECOVERY utility.

LOAD
Revokes the privilege to use the LOAD utility to load tables.

RECOVERDB
Revokes the privilege to use the RECOVER and REPORT utilities to recover table spaces and indexes.

REORG
Revokes the privilege to use the REORG utility to reorganize table spaces and indexes.

REPAIR
Revokes the privilege to use the REPAIR and DIAGNOSE utilities.

STARTDB
Revokes the privilege to issue the START DATABASE command.

STATS
Revokes the privilege to use the RUNSTATS utility to update statistics, and the CHECK utility to test whether indexes are consistent with the data they index, and the MODIFY STATISTICS utility to delete unwanted statistics history records from the corresponding catalog tables.

STOPDB
Revokes the privilege to issue the STOP DATABASE command.

ON DATABASE *database-name,...*
Identifies databases on which you are revoking the privileges. For each database you identify, you (or the indicated grantors) must have granted at least one of the specified privileges on that database to all identified users (including PUBLIC, if specified). The same database must not be identified more than once.

FROM
Refer to "REVOKE" on page 1812 for a description of the FROM clause.

BY Refer to "REVOKE" on page 1812 for a description of the BY clause.

INCLUDING DEPENDENT PRIVILEGES or NOT INCLUDING DEPENDENT PRIVILEGES
Specifies whether revoking a privilege or an authority from an authorization ID or a role also results in revoking the grants that were made by that user. The default value is based on the authority that is being revoked and the REVOKE_DEP_PRIVILEGES system parameter:

- When ACCESSCTRL, DATAACCESS, or system DBADM authority is revoked, **NOT INCLUDING DEPENDENT PRIVILEGES** is assumed and the clause must be specified on the REVOKE statement.
- When the REVOKE_DEP_PRIVILEGES system parameter is set to NO, **NOT INCLUDING DEPENDENT PRIVILEGES** is assumed and an error is returned if the statement includes **INCLUDING DEPENDENT PRIVILEGES**.

- Otherwise, **INCLUDING DEPENDENT PRIVILEGES** is assumed and the clause must be specified on the REVOKE statement.

INCLUDING DEPENDENT PRIVILEGES
Specifies that revoking a privilege or an authority from an authorization ID or a role also results in revoking dependent privileges. This means that any grants that were made by the user will continue to be revoked, until all grants in the chain have been revoked.

INCLUDING DEPENDENT PRIVILEGES cannot be specified if the system parameter REVOKE_DEP_PRIVILEGES is set to NO, which enforces the behavior to not include the dependent privileges.

NOT INCLUDING DEPENDENT PRIVILEGES
Specifies that revoking a privilege or an authority from an authorization ID or a role does not cause the grants that were made by the user to be revoked. However, for the revoked privileges, all implications of the privilege being revoked are applied. For example, if the revoked privileges were required to bind a package successfully, that package would continue to be invalidated as a result of the package owner losing these privileges. An object might be dropped if a privilege is revoked that was used to create the object.

NOT INCLUDING DEPENDENT PRIVILEGES must be specified when ACCESSCTRL, DATAACCESS, or system DBADM authority is revoked.

NOT INCLUDING DEPENDENT PRIVILEGES cannot be specified if the system parameter REVOKE_DEP_PRIVILEGES is set toYES, which enforces the behavior to include dependent privileges in the revoke.

Examples

Example 1: Revoke drop privileges on database DSN8D11A from user PEREZ.

```
REVOKE DROP
  ON DATABASE DSN8D11A
  FROM PEREZ;
```

Example 2: Revoke repair privileges on database DSN8D11A from all local users. (Grants to specific users will not be affected.)

```
REVOKE REPAIR
  ON DATABASE DSN8D11A
  FROM PUBLIC;
```

Example 3: Revoke authority to create new tables and load tables in database DSN8D11A from users WALKER, PIANKA, and FUJIMOTO.

```
REVOKE CREATETAB,LOAD
  ON DATABASE DSN8D11A
  FROM WALKER,PIANKA,FUJIMOTO;
```

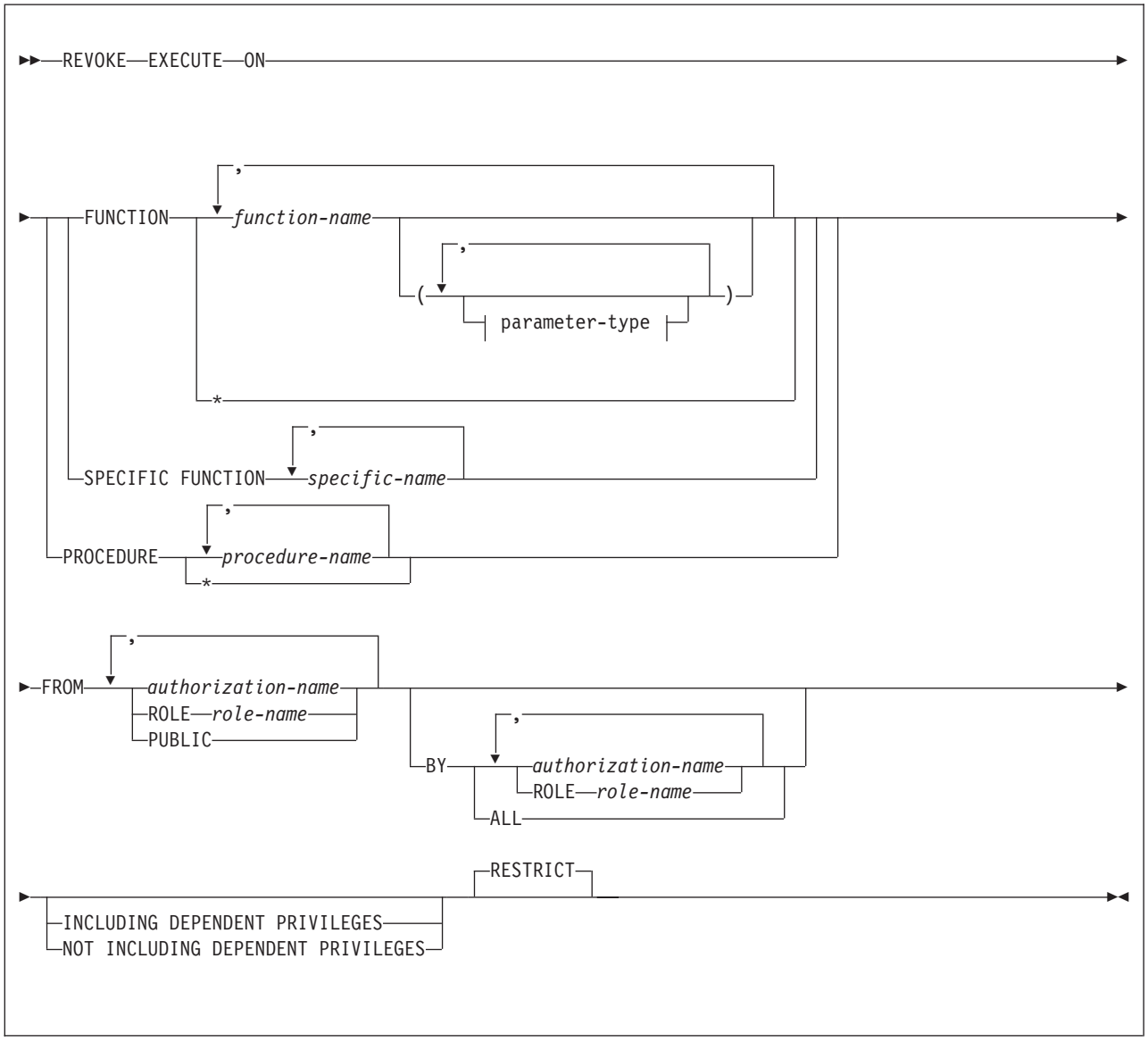
Example 4: Revoke load privileges on database DSN8D11A from role ROLE1:

```
REVOKE LOAD
  ON DATABASE DSN8D11A
  FROM ROLE ROLE1;
```

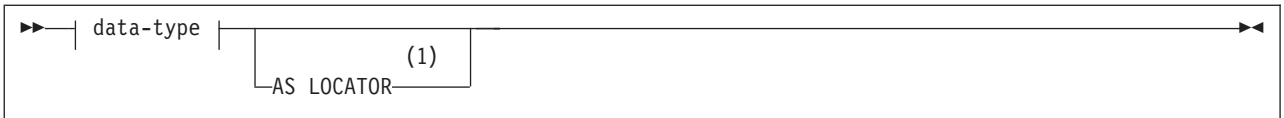
REVOKE (function or procedure privileges)

This form of the REVOKE statement revokes privileges on user-defined functions, cast functions that were generated for distinct types, and stored procedures.

Syntax



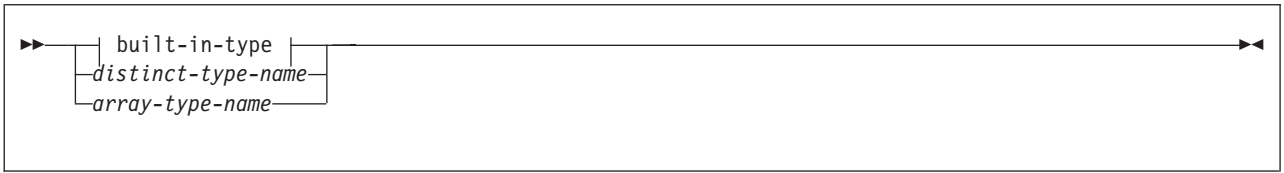
parameter-type:



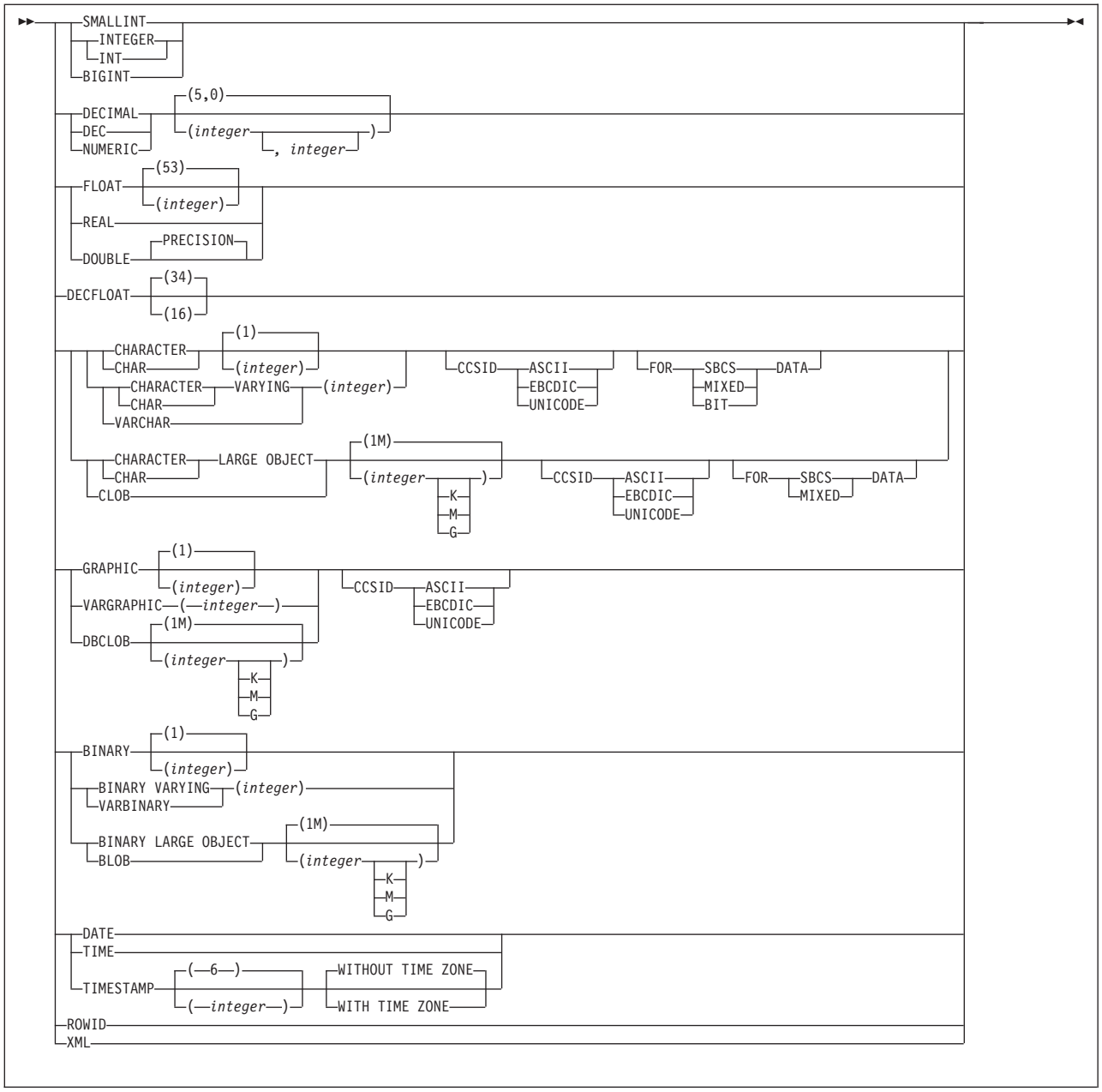
Notes:

1 AS LOCATOR can be specified only for a LOB data type or a distinct type that is based on a LOB data type.

data-type:



built-in-type:



Description

EXECUTE

Revokes the privilege to run the identified user-defined function, cast function that was generated for a distinct type, or stored procedure.

FUNCTION or SPECIFIC FUNCTION

Identifies the function from which the privilege is revoked. The function must exist at the current server, and it must be a function that was defined with the CREATE FUNCTION statement or a cast function that was generated by a CREATE TYPE statement. The function can be identified by name, function signature, or specific name.

If the function was defined with a table parameter (the LIKE TABLE was specified in the CREATE FUNCTION statement to indicate that one of the

input parameters is a transition table), the function signature cannot be used to identify the function. Instead, identify the function with its function name, if unique, or with its specific name.

FUNCTION *function-name*
Identifies the function by its name. The *function-name* must identify exactly one function. The function can have any number of parameters defined for it. If there is more than one function of the specified name in the specified or implicit schema, an error is returned.

An * can be specified for a qualified or unqualified *function-name*. An * (or *schema-name*.*) indicates that the privilege is revoked for all the functions in the schema. You (or the indicated grantors) must have granted the privilege on FUNCTION * to all identified users (including PUBLIC if specified). Privileges granted on specific functions are not affected.

FUNCTION *function-name (parameter-type,...)*
Identifies the function by its function signature, which uniquely identifies the function. The *function-name (parameter-type, ...)* must identify a function with the specified function signature. The specified parameters must match the data types in the corresponding position that were specified when the function was created. The number of data types, and the logical concatenation of the data types is used to identify the specific function instance on which the privilege is to be granted. Synonyms for data types are considered a match.

If the function was defined with a table parameter (the LIKE TABLE *name* AS LOCATOR clause was specified in the CREATE FUNCTION statement to indicate that one of the input parameters is a transition table), the function signature cannot be used to uniquely identify the function. Instead, use one of the other syntax variations to identify the function with its function name, if unique, or its specific name.

If *function-name ()* is specified, the function identified must have zero parameters.

function-name
Identifies the name of the function. If you do not explicitly qualify the function name with a schema name, the function name is implicitly qualified with a schema name as described in the preceding description for FUNCTION *function-name*.

(parameter-type,...)
Identifies the parameters of the function.

If an unqualified distinct type name is specified, DB2 searches the SQL path to resolve the schema name for the distinct type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parentheses indicate that the database manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a function defined with a data type of DEC(7,2). Similarly DECFLOAT() will be considered a match for DECFLOAT(16) or DECFLOAT(34). However, FLOAT cannot be specified with empty parenthesis because its parameter value indicates a specific data type (REAL or DOUBLE).
- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified

(implicitly or explicitly) in the CREATE FUNCTION statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).

- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

For data types with a subtype or encoding scheme attribute, specifying the FOR *subtype* DATA clause or the CCSID clause is optional. Omission of either clause indicates that DB2 ignores the attribute when determining whether the data types match. If you specify either clause, it must match the value that was implicitly or explicitly specified in the CREATE FUNCTION statement.

AS LOCATOR
Specifies that the function is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or a distinct type based on a LOB.

SPECIFIC FUNCTION *specific-name*
Identifies the function by its specific name. The *specific-name* must identify a specific function that exists at the current server.

PROCEDURE *procedure-name*
Identifies a stored procedure that is defined at the current server.

An * can be specified for a qualified or unqualified *procedure-name*. An * (or *schema-name*.*) indicates that the privilege is revoked for all the procedures in the schema. You (or the indicated grantors) must have granted the privilege on PROCEDURE * to all identified users (including PUBLIC if specified). Privileges granted on specific procedures are not affected.

FROM
Refer to "REVOKE" on page 1812 for a description of the FROM clause.

BY Refer to "REVOKE" on page 1812 for a description of the BY clause.

INCLUDING DEPENDENT PRIVILEGES or NOT INCLUDING DEPENDENT PRIVILEGES
Specifies whether revoking a privilege or an authority from an authorization ID or a role also results in revoking the grants that were made by that user. The default value is based on the authority that is being revoked and the REVOKE_DEP_PRIVILEGES system parameter:

- When ACCESSCTRL, DATAACCESS, or system DBADM authority is revoked, **NOT INCLUDING DEPENDENT PRIVILEGES** is assumed and the clause must be specified on the REVOKE statement.
- When the REVOKE_DEP_PRIVILEGES system parameter is set to NO, **NOT INCLUDING DEPENDENT PRIVILEGES** is assumed and an error is returned if the statement includes **INCLUDING DEPENDENT PRIVILEGES**.
- Otherwise, **INCLUDING DEPENDENT PRIVILEGES** is assumed and the clause must be specified on the REVOKE statement.

INCLUDING DEPENDENT PRIVILEGES
Specifies that revoking a privilege or an authority from an authorization ID or a role also results in revoking dependent privileges. This means that any

grants that were made by the user will continue to be revoked, until all grants in the chain have been revoked.

INCLUDING DEPENDENT PRIVILEGES cannot be specified if the system parameter REVOKE_DEP_PRIVILEGES is set to NO, which enforces the behavior to not include the dependent privileges.

NOT INCLUDING DEPENDENT PRIVILEGES
Specifies that revoking a privilege or an authority from an authorization ID or a role does not cause the grants that were made by the user to be revoked. However, for the revoked privileges, all implications of the privilege being revoked are applied. For example, if the revoked privileges were required to bind a package successfully, that package would continue to be invalidated as a result of the package owner losing these privileges. An object might be dropped if a privilege is revoked that was used to create the object.

NOT INCLUDING DEPENDENT PRIVILEGES must be specified when ACCESSCTRL, DATAACCESS, or system DBADM authority is revoked.

NOT INCLUDING DEPENDENT PRIVILEGES cannot be specified if the system parameter REVOKE_DEP_PRIVILEGES is set toYES, which enforces the behavior to include dependent privileges in the revoke.

RESTRICT
Prevents the EXECUTE privilege from being revoked on a user-defined function or stored procedure if the revokee owns any of the following objects and does not have the EXECUTE privilege from another source:

- A function that is sourced on the function
- A view that uses the function
- A trigger package that uses the function or stored procedure
- A table that uses the function in a check constraint or user-defined default clause
- A materialized query table whose fullselect uses the function
- An extended index that uses the function

Examples

Example 1: Revoke the EXECUTE privilege on function CALC_SALARY for user JONES. Assume that there is only one function in the schema with function CALC_SALARY.

```
REVOKE EXECUTE ON FUNCTION CALC_SALARY FROM JONES;
```

Example 2: Revoke the EXECUTE privilege on procedure VACATION_ACCR from all users at the current server.

```
REVOKE EXECUTE ON PROCEDURE VACATION_ACCR FROM PUBLIC;
```

Example 3: Revoke the privilege of the administrative assistant to grant EXECUTE privileges on function DEPT_TOTAL to other users. The administrative assistant will still have the EXECUTE privilege on function DEPT_TOTALS.

```
REVOKE EXECUTE ON FUNCTION DEPT_TOTALS
      FROM ADMIN_A;
```

Example 4: Revoke the EXECUTE privilege on function NEW_DEPT_HIRES for HR (Human Resources). The function has two input parameters with data types of INTEGER and CHAR(10), respectively. Assume that the schema has more than one function that is named NEW_DEPT_HIRES.

```
REVOKE EXECUTE ON FUNCTION NEW_DEPT_HIRES (INTEGER, CHAR(10))
      FROM HR;
```

You can also code the CHAR(10) data type as CHAR().

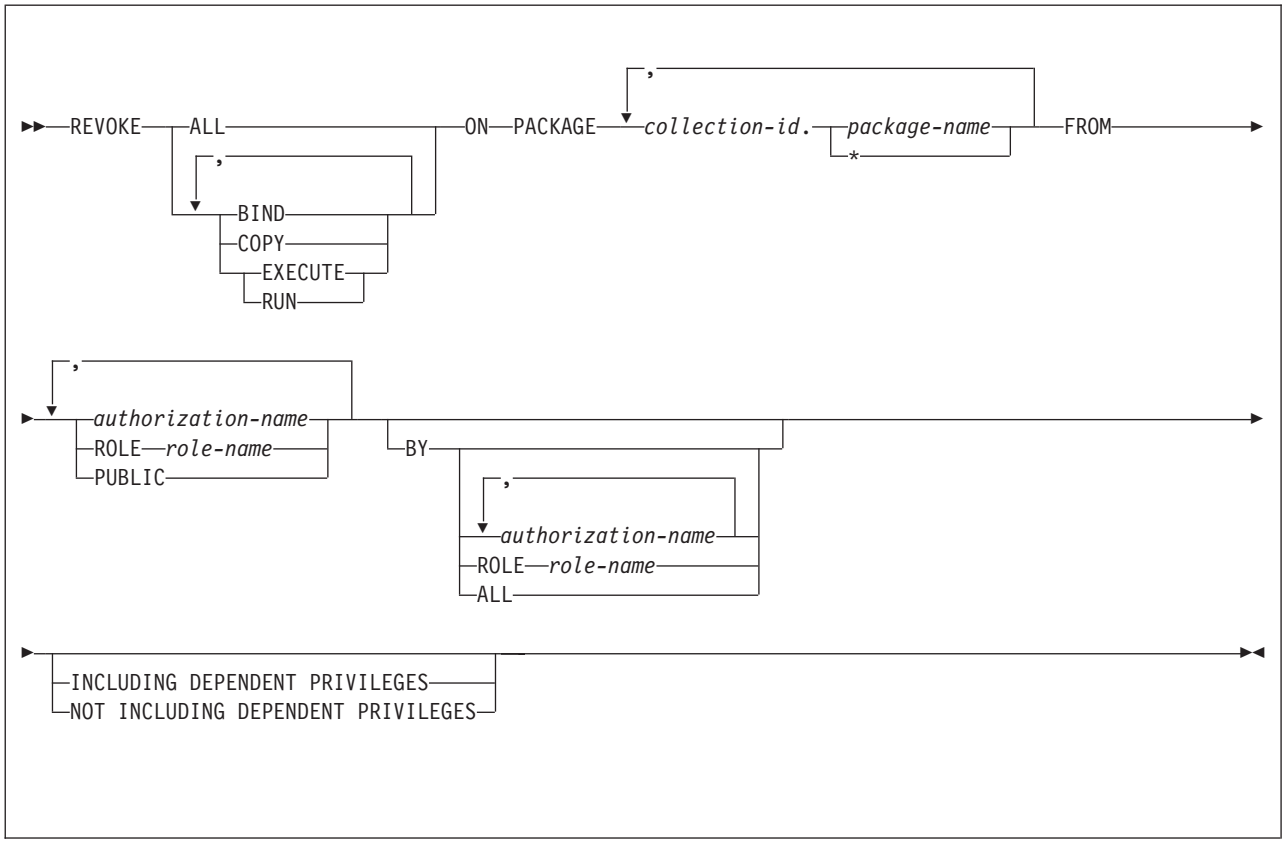
Example 5: Revoke the EXECUTE privilege on function FIND_EMPDEPT from role ROLE1:

```
REVOKE EXECUTE ON FUNCTION FIND_EMPDEPT
      FROM ROLE ROLE1;
```

REVOKE (package privileges)

This form of the REVOKE statement revokes privileges on packages.

Syntax



Description

BIND
Revokes the privilege to use the BIND and REBIND subcommands for the designated packages. In addition, if the value of field BIND NEW PACKAGE on installation panel DSNTIPP is BIND, the additional BIND privilege of adding new versions of packages is revoked. (For details, see "Notes" on page 1709 for "GRANT (package privileges)" on page 1708.)

COPY
Revokes the privilege to use the COPY option of the BIND subcommand for the designated packages.

EXECUTE
Revokes the privilege to run application programs that use the designated packages and to specify the packages following PKLIST for the BIND PLAN and REBIND PLAN commands. RUN is an alternate name for the same privilege.

ALL
Revokes all package privileges for which you have authority for the packages named in the ON clause.

ON PACKAGE *collection-id.package-name,...*
Identifies packages for which you are revoking privileges. The revoking of a package privilege applies to all versions of that package. For each package that you identify, you (or the indicated grantors) must have granted at least one of the specified privileges on that package to all identified users (including PUBLIC, if specified). An authorization ID with PACKADM authority over the collection or all collections, SYSADM, or SYSCTRL authority can specify all packages in the collection by using * for *package-name*. The same package must not be specified more than once.

FROM
Refer to "REVOKE" on page 1812 for a description of the FROM clause.

BY Refer to "REVOKE" on page 1812 for a description of the BY clause.

INCLUDING DEPENDENT PRIVILEGES or NOT INCLUDING DEPENDENT PRIVILEGES
Specifies whether revoking a privilege or an authority from an authorization ID or a role also results in revoking the grants that were made by that user. The default value is based on the authority that is being revoked and the REVOKE_DEP_PRIVILEGES system parameter:

- When ACCESSCTRL, DATAACCESS, or system DBADM authority is revoked, **NOT INCLUDING DEPENDENT PRIVILEGES** is assumed and the clause must be specified on the REVOKE statement.
- When the REVOKE_DEP_PRIVILEGES system parameter is set to NO, **NOT INCLUDING DEPENDENT PRIVILEGES** is assumed and an error is returned if the statement includes **INCLUDING DEPENDENT PRIVILEGES**.
- Otherwise, **INCLUDING DEPENDENT PRIVILEGES** is assumed and the clause must be specified on the REVOKE statement.

INCLUDING DEPENDENT PRIVILEGES
Specifies that revoking a privilege or an authority from an authorization ID or a role also results in revoking dependent privileges. This means that any grants that were made by the user will continue to be revoked, until all grants in the chain have been revoked.

INCLUDING DEPENDENT PRIVILEGES cannot be specified if the system parameter REVOKE_DEP_PRIVILEGES is set to NO, which enforces the behavior to not include the dependent privileges.

NOT INCLUDING DEPENDENT PRIVILEGES
Specifies that revoking a privilege or an authority from an authorization ID or a role does not cause the grants that were made by the user to be revoked. However, for the revoked privileges, all implications of the privilege being revoked are applied. For example, if the revoked privileges were required to bind a package successfully, that package would continue to be invalidated as a result of the package owner losing these privileges. An object might be dropped if a privilege is revoked that was used to create the object.

NOT INCLUDING DEPENDENT PRIVILEGES must be specified when ACCESSCTRL, DATAACCESS, or system DBADM authority is revoked.

NOT INCLUDING DEPENDENT PRIVILEGES cannot be specified if the system parameter REVOKE_DEP_PRIVILEGES is set toYES, which enforces the behavior to include dependent privileges in the revoke.

Notes

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports specifying PROGRAM as a synonym for PACKAGE.

Examples

Example 1: Revoke the privilege to copy all packages in collection DSN8CC61 from LEWIS.

```
REVOKE COPY ON PACKAGE DSN8CC61.* FROM LEWIS;
```

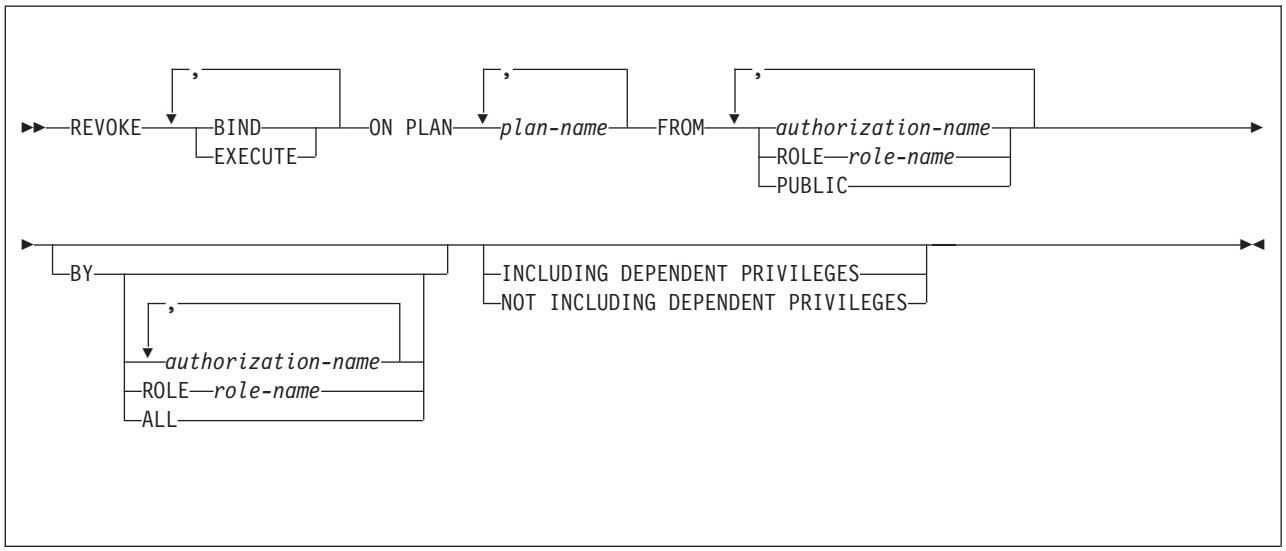
Example 2: Revoke the privilege to run all packages in collection DSN9CC13 from role ROLE1:

```
REVOKE EXECUTE ON PACKAGE DSN9CC13.* FROM ROLE ROLE1;
```

REVOKE (plan privileges)

This form of the REVOKE statement revokes privileges on application plans.

Syntax



Description

BIND
Revokes the privilege to use the BIND, REBIND, and FREE subcommands for the identified plans.

EXECUTE
Revokes the privilege to run application programs that use the identified plans.

ON PLAN *plan-name,...*
Identifies application plans for which you are revoking privileges. For each plan that you identify, you (or the indicated grantors) must have granted at least one of the specified privileges on that plan to all identified users (including PUBLIC, if specified). The same plan must not be specified more than once.

FROM
Refer to "REVOKE" on page 1812 for a description of the FROM clause.

BY Refer to "REVOKE" on page 1812 for a description of the BY clause.

INCLUDING DEPENDENT PRIVILEGES or NOT INCLUDING DEPENDENT PRIVILEGES
Specifies whether revoking a privilege or an authority from an authorization ID or a role also results in revoking the grants that were made by that user. The default value is based on the authority that is being revoked and the REVOKE_DEP_PRIVILEGES system parameter:

- When ACCESSCTRL, DATAACCESS, or system DBADM authority is revoked, **NOT INCLUDING DEPENDENT PRIVILEGES** is assumed and the clause must be specified on the REVOKE statement.
- When the REVOKE_DEP_PRIVILEGES system parameter is set to NO, **NOT INCLUDING DEPENDENT PRIVILEGES** is assumed and an error is returned if the statement includes **INCLUDING DEPENDENT PRIVILEGES**.

- Otherwise, **INCLUDING DEPENDENT PRIVILEGES** is assumed and the clause must be specified on the REVOKE statement.

INCLUDING DEPENDENT PRIVILEGES
Specifies that revoking a privilege or an authority from an authorization ID or a role also results in revoking dependent privileges. This means that any grants that were made by the user will continue to be revoked, until all grants in the chain have been revoked.

INCLUDING DEPENDENT PRIVILEGES cannot be specified if the system parameter REVOKE_DEP_PRIVILEGES is set to NO, which enforces the behavior to not include the dependent privileges.

NOT INCLUDING DEPENDENT PRIVILEGES
Specifies that revoking a privilege or an authority from an authorization ID or a role does not cause the grants that were made by the user to be revoked. However, for the revoked privileges, all implications of the privilege being revoked are applied. For example, if the revoked privileges were required to bind a package successfully, that package would continue to be invalidated as a result of the package owner losing these privileges. An object might be dropped if a privilege is revoked that was used to create the object.

NOT INCLUDING DEPENDENT PRIVILEGES must be specified when ACCESSCTRL, DATAACCESS, or system DBADM authority is revoked.

NOT INCLUDING DEPENDENT PRIVILEGES cannot be specified if the system parameter REVOKE_DEP_PRIVILEGES is set toYES, which enforces the behavior to include dependent privileges in the revoke.

Examples

Example 1: Revoke authority to bind plan DSN8IP11 from user JONES.

```
REVOKE BIND ON PLAN DSN8IP11 FROM JONES;
```

Example 2: Revoke authority previously granted to all users at the current server to bind and execute plan DSN8CP11. (Grants to specific users will not be affected.)

```
REVOKE BIND,EXECUTE ON PLAN DSN8CP11 FROM PUBLIC;
```

Example 3: Revoke authority to execute plan DSN8CP11 from users ADAMSON and BROWN.

```
REVOKE EXECUTE ON PLAN DSN8CP11 FROM ADAMSON,BROWN;
```

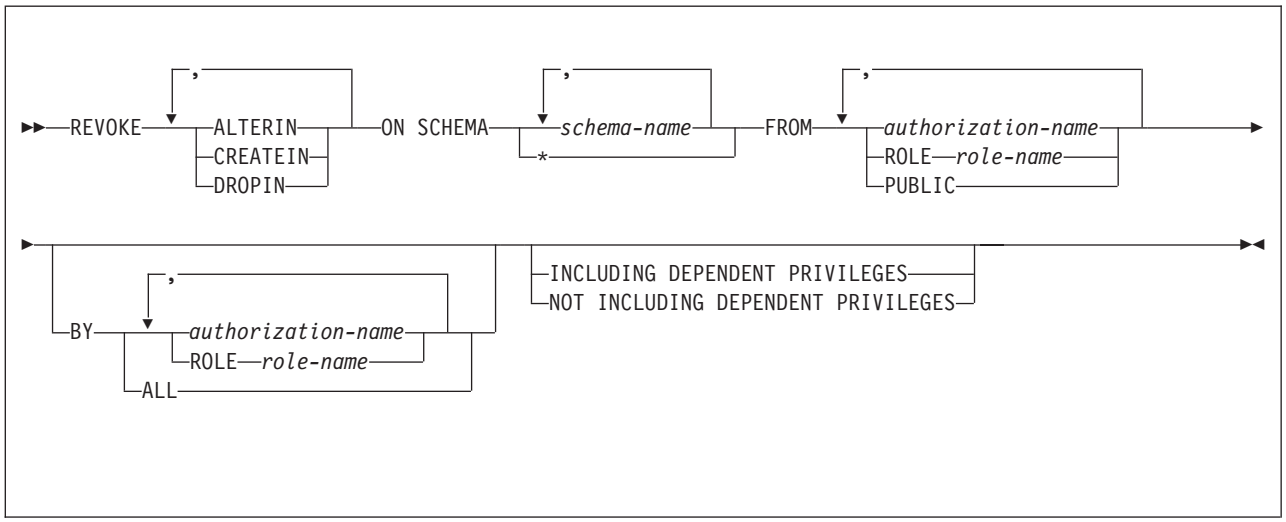
Example 4: Revoke authority to bind plan DSN91PLN from role ROLE1:

```
REVOKE BIND ON PLAN DSN91PLN FROM ROLE ROLE1;
```

REVOKE (schema privileges)

This form of the REVOKE statement revokes privileges on schemas.

Syntax



Description

ALTERIN
Revokes the privilege to alter sequences, stored procedures, and user-defined functions, or specify a comment for distinct types, cast functions that are generated for distinct types, sequences, stored procedures, triggers, and user-defined functions in the designated schemas.

CREATEIN
Revokes the privilege to create distinct types, sequences, stored procedures, triggers, and user-defined functions in the designated schemas.

DROPIN
Revokes the privilege to drop distinct types, sequences, stored procedures, triggers, and user-defined functions in the designated schemas.

SCHEMA *schema-name*
Identifies the schema on which the privilege is revoked.

SCHEMA *
Indicates that the specified privilege on all schemas is revoked. You (or the indicated grantors) must have previously granted the specified privilege on SCHEMA * to all identified users (including PUBLIC if specified). Privileges granted on specific schemas are not affected.

FROM
Refer to "REVOKE" on page 1812 for a description of the FROM clause.

BY Refer to "REVOKE" on page 1812 for a description of the BY clause.

INCLUDING DEPENDENT PRIVILEGES or NOT INCLUDING DEPENDENT PRIVILEGES
Specifies whether revoking a privilege or an authority from an authorization ID or a role also results in revoking the grants that were made by that user. The default value is based on the authority that is being revoked and the REVOKE_DEP_PRIVILEGES system parameter:

- When ACCESSCTRL, DATAACCESS, or system DBADM authority is revoked, **NOT INCLUDING DEPENDENT PRIVILEGES** is assumed and the clause must be specified on the REVOKE statement.
- When the REVOKE_DEP_PRIVILEGES system parameter is set to NO, **NOT INCLUDING DEPENDENT PRIVILEGES** is assumed and an error is returned if the statement includes **INCLUDING DEPENDENT PRIVILEGES**.
- Otherwise, **INCLUDING DEPENDENT PRIVILEGES** is assumed and the clause must be specified on the REVOKE statement.

INCLUDING DEPENDENT PRIVILEGES
Specifies that revoking a privilege or an authority from an authorization ID or a role also results in revoking dependent privileges. This means that any grants that were made by the user will continue to be revoked, until all grants in the chain have been revoked.

INCLUDING DEPENDENT PRIVILEGES cannot be specified if the system parameter REVOKE_DEP_PRIVILEGES is set to NO, which enforces the behavior to not include the dependent privileges.

NOT INCLUDING DEPENDENT PRIVILEGES
Specifies that revoking a privilege or an authority from an authorization ID or a role does not cause the grants that were made by the user to be revoked. However, for the revoked privileges, all implications of the privilege being revoked are applied. For example, if the revoked privileges were required to bind a package successfully, that package would continue to be invalidated as a result of the package owner losing these privileges. An object might be dropped if a privilege is revoked that was used to create the object.

NOT INCLUDING DEPENDENT PRIVILEGES must be specified when ACCESSCTRL, DATAACCESS, or system DBADM authority is revoked.

NOT INCLUDING DEPENDENT PRIVILEGES cannot be specified if the system parameter REVOKE_DEP_PRIVILEGES is set toYES, which enforces the behavior to include dependent privileges in the revoke.

Examples

Example 1: Revoke the CREATEIN privilege on schema T_SCORES from user JONES.

```
REVOKE CREATEIN ON SCHEMA T_SCORES FROM JONES;
```

Example 2: Revoke the CREATEIN privilege on schema VAC from all users at the current server.

```
REVOKE CREATEIN ON SCHEMA VAC FROM PUBLIC;
```

Example 3: Revoke the ALTERIN privilege on schema DEPT from the administrative assistant.

```
REVOKE ALTERIN ON SCHEMA DEPT FROM ADMIN_A;
```

Example 4: Revoke the ALTERIN and DROPIN privileges on schemas NEW_HIRE, PROMO, and RESIGN from HR (Human Resources).

```
REVOKE ALTERIN, DROPIN ON SCHEMA NEW_HIRE, PROMO, RESIGN FROM HR;
```

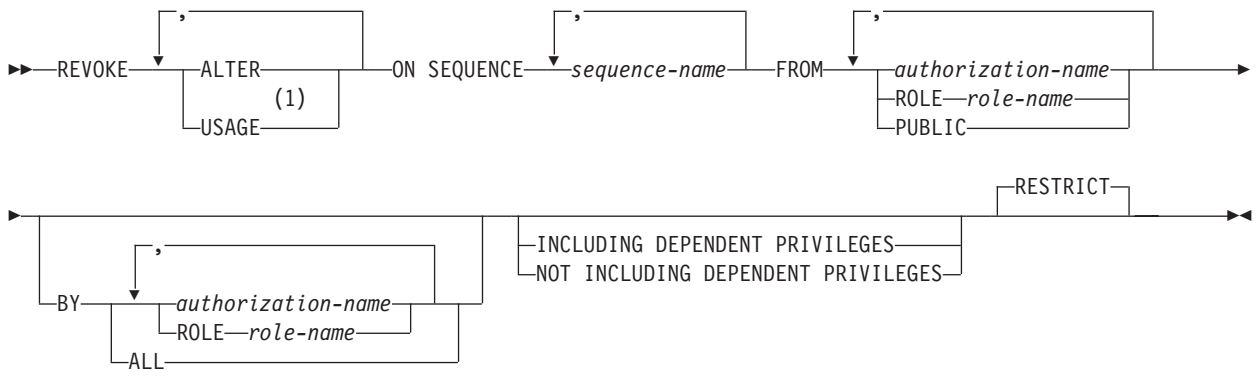
Example 5: Revoke the ALTERIN privilege on schemas EMPLOYEE from role ROLE1:

```
REVOKE ALTERIN ON SCHEMA EMPLOYEE FROM ROLE ROLE1;
```

REVOKE (sequence privileges)

This form of the REVOKE statement revokes the privileges on a user-defined sequence.

Syntax



Notes:

1 The keyword SELECT is an alternative keyword for USAGE.

Description

ALTER
Revokes the privilege to alter a sequence or record a comment on a sequence.

USAGE
Revokes the USAGE privilege to use a sequence. This privilege is needed when the NEXT VALUE or PREVIOUS VALUE expression is invoked for a sequence name.

SEQUENCE *sequence-name*
Identifies the sequence. The name, including the implicit or explicit schema qualifier, must uniquely identify an existing sequence at the current server. If no sequence by this name exists in the explicitly or implicitly specified schema, an error occurs. *sequence-name* must not be the name of an internal sequence object that is generated by the system for an identity column.

FROM
Refer to "REVOKE" on page 1812 for a description of the FROM clause.

BY Refer to "REVOKE" on page 1812 for a description of the BY clause.

INCLUDING DEPENDENT PRIVILEGES or NOT INCLUDING DEPENDENT PRIVILEGES
Specifies whether revoking a privilege or an authority from an authorization ID or a role also results in revoking the grants that were made by that user. The default value is based on the authority that is being revoked and the REVOKE_DEP_PRIVILEGES system parameter:

- When ACCESSCTRL, DATAACCESS, or system DBADM authority is revoked, **NOT INCLUDING DEPENDENT PRIVILEGES** is assumed and the clause must be specified on the REVOKE statement.

- When the REVOKE_DEP_PRIVILEGES system parameter is set to NO, **NOT INCLUDING DEPENDENT PRIVILEGES** is assumed and an error is returned if the statement includes **INCLUDING DEPENDENT PRIVILEGES**.
- Otherwise, **INCLUDING DEPENDENT PRIVILEGES** is assumed and the clause must be specified on the REVOKE statement.

INCLUDING DEPENDENT PRIVILEGES
Specifies that revoking a privilege or an authority from an authorization ID or a role also results in revoking dependent privileges. This means that any grants that were made by the user will continue to be revoked, until all grants in the chain have been revoked.

INCLUDING DEPENDENT PRIVILEGES cannot be specified if the system parameter REVOKE_DEP_PRIVILEGES is set to NO, which enforces the behavior to not include the dependent privileges.

NOT INCLUDING DEPENDENT PRIVILEGES
Specifies that revoking a privilege or an authority from an authorization ID or a role does not cause the grants that were made by the user to be revoked. However, for the revoked privileges, all implications of the privilege being revoked are applied. For example, if the revoked privileges were required to bind a package successfully, that package would continue to be invalidated as a result of the package owner losing these privileges. An object might be dropped if a privilege is revoked that was used to create the object.

NOT INCLUDING DEPENDENT PRIVILEGES must be specified when ACCESSCTRL, DATAACCESS, or system DBADM authority is revoked.

NOT INCLUDING DEPENDENT PRIVILEGES cannot be specified if the system parameter REVOKE_DEP_PRIVILEGES is set toYES, which enforces the behavior to include dependent privileges in the revoke.

RESTRICT
Prevents the USAGE privilege from being revoked on a sequence if the revokee owns one of the following objects and does not have the USAGE privilege from another source:

- A trigger that specifies the sequence in a NEXT VALUE or PREVIOUS VALUE expression
- An inline SQL function that specifies the sequence in a NEXT VALUE or PREVIOUS VALUE expression

Examples

Example 1: Revoke USAGE privilege on sequence MYNUM to user JONES.

```
REVOKE USAGE
  ON SEQUENCE MYNUM
  FROM JONES;
```

Example 2: Revoke the USAGE privilege on sequence ORDER_SEQ from role ROLE1:

```
REVOKE USAGE
  ON SEQUENCE ORDER_SEQ
  FROM ROLE ROLE1;
```

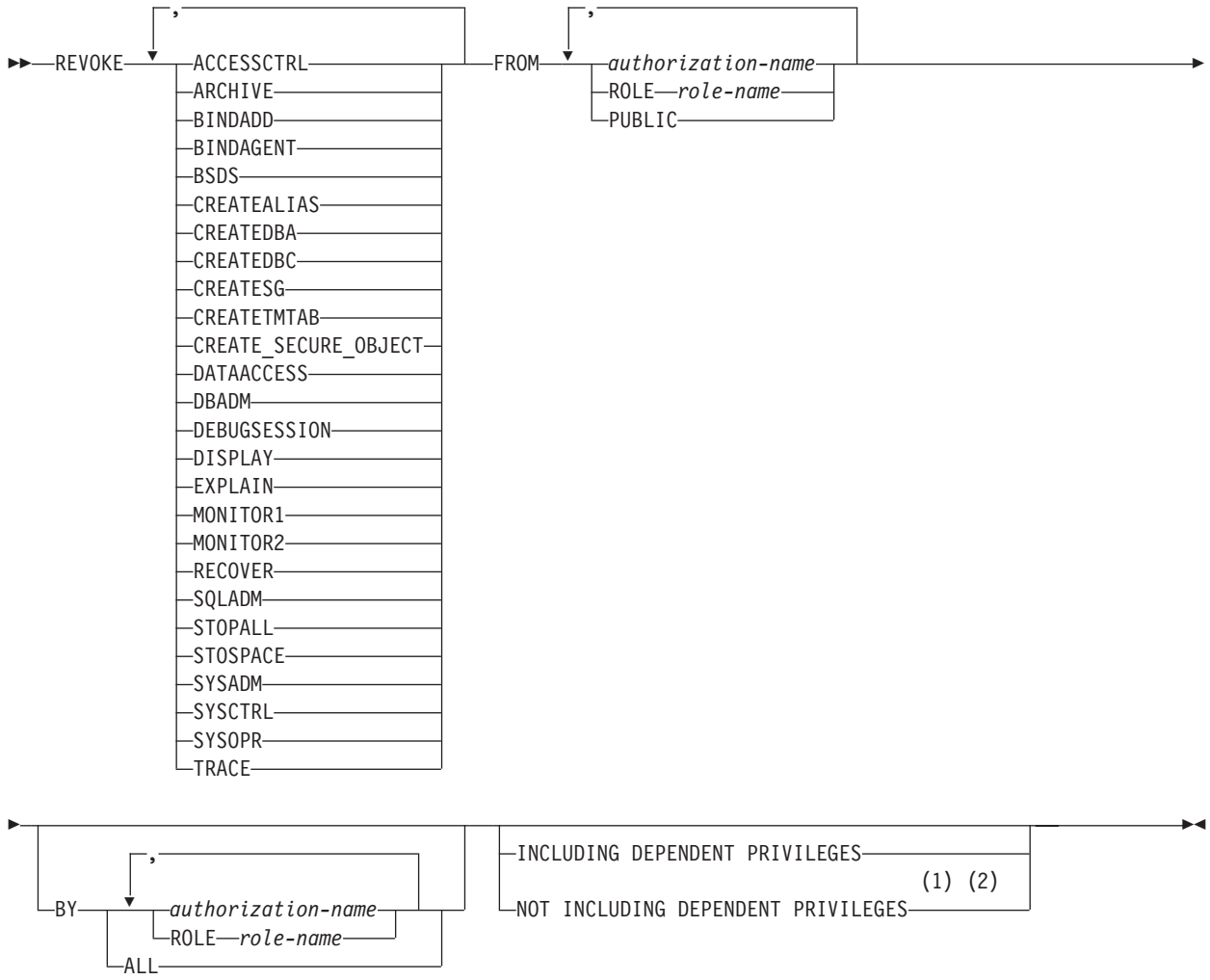
REVOKE (system privileges)

This form of the REVOKE statement revokes system privileges.

Authorization

To revoke the CREATE_SECURE_OBJECT privilege with or without the BY clause, the privilege set must include SECADM authority.

Syntax



Notes:

1 **INCLUDING DEPENDENT PRIVILEGES** must not be specified when **ACCESSCTRL**, **DATAACCESS**, or **DBADM** is specified.

2 **NOT INCLUDING DEPENDENT PRIVILEGES** must be specified when **ACCESSCTRL**, **DATAACCESS**, or **DBADM** is specified

Description

ACCESSCTRL
Revokes the ACCESSCTRL authority, but does not revoke any privileges that are dependent on it.

ARCHIVE
Revokes the privilege to use the ARCHIVE LOG command.

BINDADD
Revokes the privilege to create plans and packages using the BIND subcommand with the ADD option.

BINDAGENT
Revokes the privilege to issue the BIND, FREE PACKAGE, or REBIND subcommands for plans and packages and the DROP PACKAGE statement on behalf of the grantor. The privilege also allows the holder to copy and replace plans and packages on behalf of the grantor.

A revoke of this privilege does not cascade.

BSDS
Revokes the privilege to issue the RECOVER BSDS command.

CREATEALIAS
Revokes the privilege to use the CREATE ALIAS statement.

CREATEDBA
Revokes the privilege to issue the CREATE DATABASE statement and acquire DBADM authority over those databases.

CREATEDBC
Revokes the privilege to issue the CREATE DATABASE statement and acquire DBCTRL authority over those databases.

CREATESG
Revokes the privilege to create new storage groups.

CREATETMTAB
Revokes the privilege to use the CREATE GLOBAL TEMPORARY TABLE statement.

CREATE_SECURE_OBJECT
Revokes the privilege to create a secure object.

DATAACCESS
Revokes the DATAACCESS authority, but does not revoke any privileges that are dependent on it. Revoking DATAACCESS can result in authorization cache entries (plan, package, routine, and dynamic statement) being updated if they were dependent on it. The RESTRICT semantics on objects prevents the DATAACCESS authority from being revoked if the revokee owns an object that was created with dependencies on the authority to be revoked.

Revoking DATAACCESS is similar to revoking the individual privileges that DATAACCESS includes. For example, if a view was created based on the view owner having the SELECT privilege as acquired through the DATAACCESS authority, revoking DATAACCESS would be the equivalent of revoking the SELECT privilege and the view would be dropped.

DBADM
Revokes the DBADM authority from the user. If this user was also granted DATACCESS or ACCESSCTRL authority along with DBADM authority, DATACCESS or ACCESSCTRL would not be revoked.

DISPLAY
Revokes the privilege to use the following commands:

- The DISPLAY ARCHIVE command for archive log information
- The DISPLAY BUFFERPOOL command for the status of buffer pools
- The DISPLAY DATABASE command for the status of all databases
- The DISPLAY FUNCTION SPECIFIC command for statistics about accessed external user-defined functions
- The DISPLAY LOCATION command for statistics about threads with a distributed relationship
- The DISPLAY PROCEDURE command for statistics about accessed stored procedures
- The DISPLAY THREAD command for information on active threads with in DB2
- The DISPLAY TRACE command for a list of active traces

DEBUGSESSION
Revokes the privilege to create a debug session, which prevents client application debugging of native SQL or Java procedures that are executed within the session.

EXPLAIN
Revokes the privilege to issue the following:

- The EXPLAIN statement with the following options:
  - PLAN
  - ALL
- The PREPARE statement
- The DESCRIBE TABLE statement
- The ability to explain dynamic SQL statements that are executing with the special register CURRENT EXPLAIN MODE = EXPLAIN
- The BIND options EXPLAIN(ONLY) and SQLERROR(CHECK)

  EXPLAIN(ONLY) allows to explain the statements. SQLERROR(CHECK) performs all syntax and semantic checks on the SQL statements being bound.

MONITOR1
Revokes the privilege to obtain IFC data classified as serviceability data, statistics, accounting, and other performance data that does not contain potentially secure data.

MONITOR2
Revokes the privilege to obtain IFC data classified as containing potentially sensitive data such as SQL statement text and audit data. (Having the MONITOR2 privilege also implies having MONITOR1 privileges, however, revoking the MONITOR2 privilege does not cause the revoke of an explicitly granted MONITOR1 privilege.)

RECOVER
Revokes the privilege to issue the RECOVER INDOUBT command.

SQLADM
Revokes the privilege to issue the following:

- The DESCRIBE TABLE statement
- The EXPLAIN statement with the following options:
  - PLAN

  - ALL
  - STMTCACHE ALL
  - STMTID
  - STMTTOKEN
  - MONITORED STMTS
- The PREPARE statement
- The ability to explain dynamic SQL statements that are executing with the special register CURRENT EXPLAIN MODE = EXPLAIN
- The BIND options EXPLAIN(ONLY) and SQLERROR(CHECK)

  EXPLAIN(ONLY) allows to explain the statements. SQLERROR(CHECK) performs all syntax and semantic checks on the SQL statements being bound.
- The START command
- The STOP command
- The DISPLAY PROFILE command
- The ability to execute the RUNSTATS utility and the MODIFY STATISTICS utility in any database
- MONITOR2 privilege, which allows users to obtain IFC data that is classified as containing potentially sensitive data, such as SQL statement text and audit data, as well as IFC data that is classified as serviceability data, statistics, accounting, and other performance data.

STOPALL
Revokes the privilege to use the STOP DB2 command.

STOSPACE
Revokes the privilege to use the STOSPACE utility.

SYSADM
Revokes the system administrator authority.

SYSCTRL
Revokes the system control authority.

SYSOPR
Revokes the system operator authority.

TRACE
Revokes the privilege to use the MODIFY TRACE, START TRACE, and STOP TRACE commands.

FROM
Refer to "REVOKE" on page 1812 for a description of the FROM clause.

BY Refer to "REVOKE" on page 1812 for a description of the BY clause.

INCLUDING DEPENDENT PRIVILEGES or NOT INCLUDING DEPENDENT PRIVILEGES
Specifies whether revoking a privilege or an authority from an authorization ID or a role also results in revoking the grants that were made by that user. The default value is based on the authority that is being revoked and the REVOKE_DEP_PRIVILEGES system parameter:

- When ACCESSCTRL, DATAACCESS, or system DBADM authority is revoked, **NOT INCLUDING DEPENDENT PRIVILEGES** is assumed and the clause must be specified on the REVOKE statement.

- When the REVOKE_DEP_PRIVILEGES system parameter is set to NO, **NOT INCLUDING DEPENDENT PRIVILEGES** is assumed and an error is returned if the statement includes **INCLUDING DEPENDENT PRIVILEGES**.
- Otherwise, **INCLUDING DEPENDENT PRIVILEGES** is assumed and the clause must be specified on the REVOKE statement.

INCLUDING DEPENDENT PRIVILEGES
Specifies that revoking a privilege or an authority from an authorization ID or a role also results in revoking dependent privileges. This means that any grants that were made by the user will continue to be revoked, until all grants in the chain have been revoked.

INCLUDING DEPENDENT PRIVILEGES cannot be specified if the system parameter REVOKE_DEP_PRIVILEGES is set to NO, which enforces the behavior to not include the dependent privileges.

NOT INCLUDING DEPENDENT PRIVILEGES
Specifies that revoking a privilege or an authority from an authorization ID or a role does not cause the grants that were made by the user to be revoked. However, for the revoked privileges, all implications of the privilege being revoked are applied. For example, if the revoked privileges were required to bind a package successfully, that package would continue to be invalidated as a result of the package owner losing these privileges. An object might be dropped if a privilege is revoked that was used to create the object.

NOT INCLUDING DEPENDENT PRIVILEGES must be specified when ACCESSCTRL, DATAACCESS, or system DBADM authority is revoked.

NOT INCLUDING DEPENDENT PRIVILEGES cannot be specified if the system parameter REVOKE_DEP_PRIVILEGES is set toYES, which enforces the behavior to include dependent privileges in the revoke.

Examples

Example 1: Revoke DISPLAY privileges from user LUTZ.

```
REVOKE DISPLAY
  FROM LUTZ;
```

Example 2: Revoke BSDS and RECOVER privileges from users PARKER and SETRIGHT.

```
REVOKE BSDS,RECOVER
  FROM PARKER,SETRIGHT;
```

Example 3: Revoke TRACE privileges previously granted to all local users. (Grants to specific users will not be affected.)

```
REVOKE TRACE
  FROM PUBLIC;
```

Example 4: Revoke ARCHIVE privileges from role ROLE1:

```
REVOKE ARCHIVE
  FROM ROLE ROLE1;
```

Example 5: SECADM Mary revokes the privilege to create a secure object from Steve that was granted by another SECADM.

```
REVOKE CREATE_SECURE_OBJECT
FROM STEVE BY MARY;
```

Example 6: Revoke system DBADM from the role, ADMINROLE. This only revokes system DBADM authority from the role. If DATAACCESS and ACCESSCTRL authorities were granted during GRANT DBADM, those authorities are not revoked.

```
 REVOKE DBADM ON SYSTEM
FROM ROLE ADMINROLE
NOT INCLUDING DEPENDENT PRIVILEGES;
```

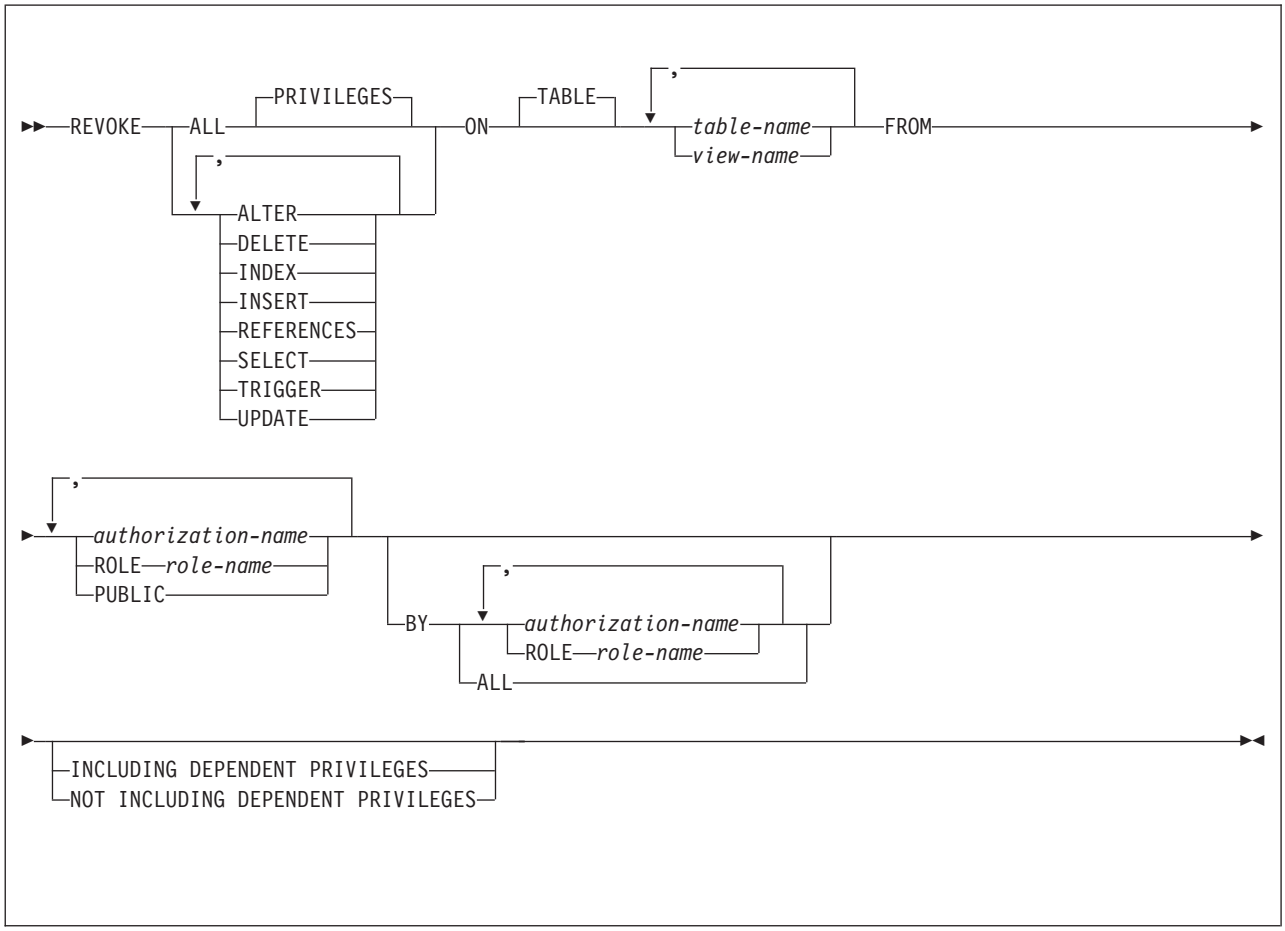
Example 7: Revoke system DBADM, DATAACCESS, and ACCESSCTRL authorities from the role, ADMINROLE.

```
 REVOKE DBADM, DATAACCESS, ACCESSCTRL ON SYSTEM
FROM ROLE ADMINROLE
NOT INCLUDING DEPENDENT PRIVILEGES;
```

REVOKE (table or view privileges)

This form of the REVOKE statement revokes privileges on one or more tables or views.

Syntax



Description

ALL or ALL PRIVILEGES

If you specify ALL, the authorization ID of the statement must have granted a least one privilege on each identified table or view to each *authorization-name*. The privilege revoked from an authorization ID are those privileges on the table or view that the authorization ID of the statement granted to the authorization ID.

If you do not use ALL, you must use one or more of the keywords listed below. Each keyword revokes the privilege described, but only as it applies to the tables or views named in the ON clause.

ALTER

Revokes the privilege to alter the specified table or create a trigger on the specified table.

DELETE

Revokes the privilege to delete rows in the specified table or view.

INDEX
Revokes he privilege to create an index on the specified table.

INSERT
Revokes the privilege to insert rows into the specified table or view.

REFERENCES
Revokes the privilege to define and drop referential constraints. Although you can use a list of column names with the GRANT statement, you cannot use a list of column names with REVOKE; the privilege is revoked for all columns.

SELECT
Revokes the privilege to create a view or read data from the specified table or view. A view or a materialized query table is dropped when the SELECT privilege that was used to create it is revoked, unless the owner of the view or materialized query table was directly granted the SELECT privilege from another source before the view or materialized query table was created.

TRIGGER
Revokes the privilege to create a trigger on the specified table.

UPDATE
Revokes the privilege to update rows in the specified table or view. A list of column names can be used only with GRANT, not with REVOKE.

ON *table-name* **or** *view-name*
Names one or more tables or views on which you are revoking the privileges. The list can consist of table names, view names, or a combination of the two. A table or view must not be identified more than one time, and a declared temporary table and a table that is implicitly created for an XML column must not be identified.

FROM
Refer to "REVOKE" on page 1812 for a description of the FROM clause.

BY If you omit BY, you must have granted each named privilege to each of the named users. More precisely, each privilege must have been granted to each user by a GRANT statement whose authorization ID is also the authorization ID of your REVOKE statement. Each of these grants is then revoked. (No single privilege need be granted on all tables and views.)

If BY is specified, each named grantor must satisfy the above requirement. In that case, the authorization ID of the statement need not satisfy the requirement unless it is one of the named grantors.

Refer to "REVOKE" on page 1812 for a description of the BY clause.

INCLUDING DEPENDENT PRIVILEGES or NOT INCLUDING DEPENDENT PRIVILEGES
Specifies whether revoking a privilege or an authority from an authorization ID or a role also results in revoking the grants that were made by that user. The default value is based on the authority that is being revoked and the REVOKE_DEP_PRIVILEGES system parameter:

- When ACCESSCTRL, DATAACCESS, or system DBADM authority is revoked, **NOT INCLUDING DEPENDENT PRIVILEGES** is assumed and the clause must be specified on the REVOKE statement.
- When the REVOKE_DEP_PRIVILEGES system parameter is set to NO, **NOT INCLUDING DEPENDENT PRIVILEGES** is assumed and an error is returned if the statement includes **INCLUDING DEPENDENT PRIVILEGES**.
- Otherwise, **INCLUDING DEPENDENT PRIVILEGES** is assumed and the clause must be specified on the REVOKE statement.

INCLUDING DEPENDENT PRIVILEGES
Specifies that revoking a privilege or an authority from an authorization ID or a role also results in revoking dependent privileges. This means that any grants that were made by the user will continue to be revoked, until all grants in the chain have been revoked.

INCLUDING DEPENDENT PRIVILEGES cannot be specified if the system parameter REVOKE_DEP_PRIVILEGES is set to NO, which enforces the behavior to not include the dependent privileges.

NOT INCLUDING DEPENDENT PRIVILEGES
Specifies that revoking a privilege or an authority from an authorization ID or a role does not cause the grants that were made by the user to be revoked. However, for the revoked privileges, all implications of the privilege being revoked are applied. For example, if the revoked privileges were required to bind a package successfully, that package would continue to be invalidated as a result of the package owner losing these privileges. An object might be dropped if a privilege is revoked that was used to create the object.

NOT INCLUDING DEPENDENT PRIVILEGES must be specified when ACCESSCTRL, DATAACCESS, or system DBADM authority is revoked.

NOT INCLUDING DEPENDENT PRIVILEGES cannot be specified if the system parameter REVOKE_DEP_PRIVILEGES is set toYES, which enforces the behavior to include dependent privileges in the revoke.

Notes

For a created temporary table, only ALL or ALL PRIVILEGES can be revoked. Specific table privileges cannot be revoked.

For a view of a created temporary table, either ALL or the specific UPDATE, DELETE, INSERT and SELECT privileges can be revoked.

For a declared temporary table, no privileges can be revoked because none can be granted. When a declared temporary table is defined, PUBLIC implicitly receives all table privileges (without GRANT authority) for the table. These privileges are not recorded in the DB2 catalog.

Examples

Example 1: Revoke SELECT privileges on table DSN8B10.EMP from user PULASKI.

```
REVOKE SELECT ON TABLE DSN8B10.EMP FROM PULASKI;
```

Example 2: Revoke update privileges on table DSN8B10.EMP previously granted to all local DB2 users. (Grants to specific users are not affected.)

```
REVOKE UPDATE ON TABLE DSN8B10.EMP FROM PUBLIC;
```

Example 3: Revoke all privileges on table DSN8B10.EMP from users KWAN and THOMPSON.

```
REVOKE ALL ON TABLE DSN8B10.EMP FROM KWAN,THOMPSON;
```

Example 4: Revoke the grant of SELECT and UPDATE privileges on the table DSN8B10.DEPT to every user in the network. Doing so does not affect users who obtained these privileges from some other grant.

```
REVOKE SELECT, UPDATE ON TABLE DSN8B10.DEPT
  FROM PUBLIC;
```

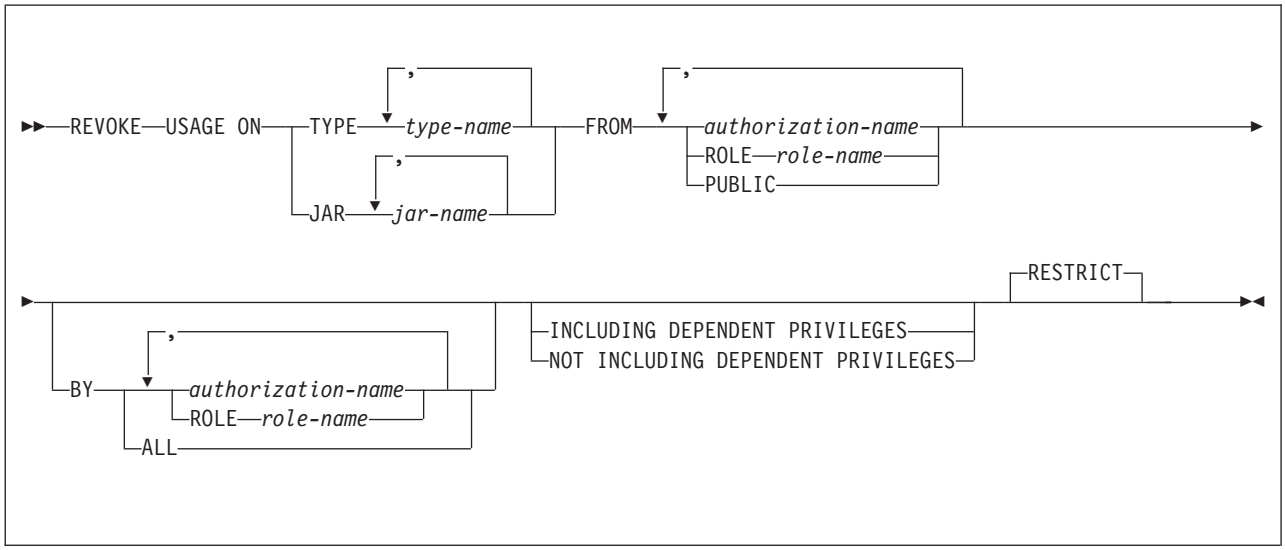
Example 5: Revoke the ALTER privileges on the table DSN8B10.EMP that were previously granted to role ROLE1:

```
REVOKE ALTER ON TABLE DSN8B10.EMP
  FROM ROLE ROLE1;
```

REVOKE (type or JAR file privileges)

This form of the REVOKE statement revokes the privilege to use distinct types, array types, or JAR files.

Syntax



Description

USAGE
Revokes the privilege to use the distinct type in tables, functions procedures, or the privilege to use the JAR file.

TYPE *type-name*
Identifies the user-defined type. The name, including the implicit or explicit schema name, must identify a unique user-defined type that exists at the current server.

JAR *jar-name*
Identifies the JAR file. The name, including the implicit or explicit schema name, must identify a unique JAR file that exists at the current server.

FROM
Refer to "REVOKE" on page 1812 for a description of the FROM clause.

BY Refer to "REVOKE" on page 1812 for a description of the BY clause.

INCLUDING DEPENDENT PRIVILEGES or NOT INCLUDING DEPENDENT PRIVILEGES
Specifies whether revoking a privilege or an authority from an authorization ID or a role also results in revoking the grants that were made by that user. The default value is based on the authority that is being revoked and the REVOKE_DEP_PRIVILEGES system parameter:

- When ACCESSCTRL, DATAACCESS, or system DBADM authority is revoked, **NOT INCLUDING DEPENDENT PRIVILEGES** is assumed and the clause must be specified on the REVOKE statement.
- When the REVOKE_DEP_PRIVILEGES system parameter is set to NO, **NOT INCLUDING DEPENDENT PRIVILEGES** is assumed and an error is returned if the statement includes **INCLUDING DEPENDENT PRIVILEGES**.

- Otherwise, **INCLUDING DEPENDENT PRIVILEGES** is assumed and the clause must be specified on the REVOKE statement.

INCLUDING DEPENDENT PRIVILEGES
Specifies that revoking a privilege or an authority from an authorization ID or a role also results in revoking dependent privileges. This means that any grants that were made by the user will continue to be revoked, until all grants in the chain have been revoked.

INCLUDING DEPENDENT PRIVILEGES cannot be specified if the system parameter REVOKE_DEP_PRIVILEGES is set to NO, which enforces the behavior to not include the dependent privileges.

NOT INCLUDING DEPENDENT PRIVILEGES
Specifies that revoking a privilege or an authority from an authorization ID or a role does not cause the grants that were made by the user to be revoked. However, for the revoked privileges, all implications of the privilege being revoked are applied. For example, if the revoked privileges were required to bind a package successfully, that package would continue to be invalidated as a result of the package owner losing these privileges. An object might be dropped if a privilege is revoked that was used to create the object.

NOT INCLUDING DEPENDENT PRIVILEGES must be specified when ACCESSCTRL, DATAACCESS, or system DBADM authority is revoked.

NOT INCLUDING DEPENDENT PRIVILEGES cannot be specified if the system parameter REVOKE_DEP_PRIVILEGES is set toYES, which enforces the behavior to include dependent privileges in the revoke.

RESTRICT
Prevents the USAGE privilege from being revoked on a user-defined type or JAR file if any of the following conditions exist and the revokee does not have the USAGE privilege from another source:

- The revokee owns a function or stored procedure that uses the user-defined type or references the JAR file.
- The revokee owns a JAR file whose path references the JAR file for which USAGE is being revoked.
- The revokee owns a table that has a column that uses the user-defined type.
- A sequence exists for which the data type of the sequence is the user-defined type.

Notes

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports DATA TYPE or DISTINCT TYPE as a synonym for TYPE.

Examples

Example 1: Revoke the USAGE privilege on distinct type SHOESIZE from user JONES.

```
REVOKE USAGE ON TYPE SHOESIZE FROM JONES;
```

Example 2: Revoke the USAGE privilege on distinct type US_DOLLAR from all users at the current server except for those who have been specifically granted USAGE and not through PUBLIC.

```
REVOKE USAGE ON TYPE US_DOLLAR FROM PUBLIC;
```

Example 3: Revoke the USAGE privilege on distinct type CANADIAN_DOLLARS from the administrative assistant (ADMIN_A).

```
REVOKE USAGE ON TYPE CANADIAN_DOLLARS
      FROM ADMIN_A;
```

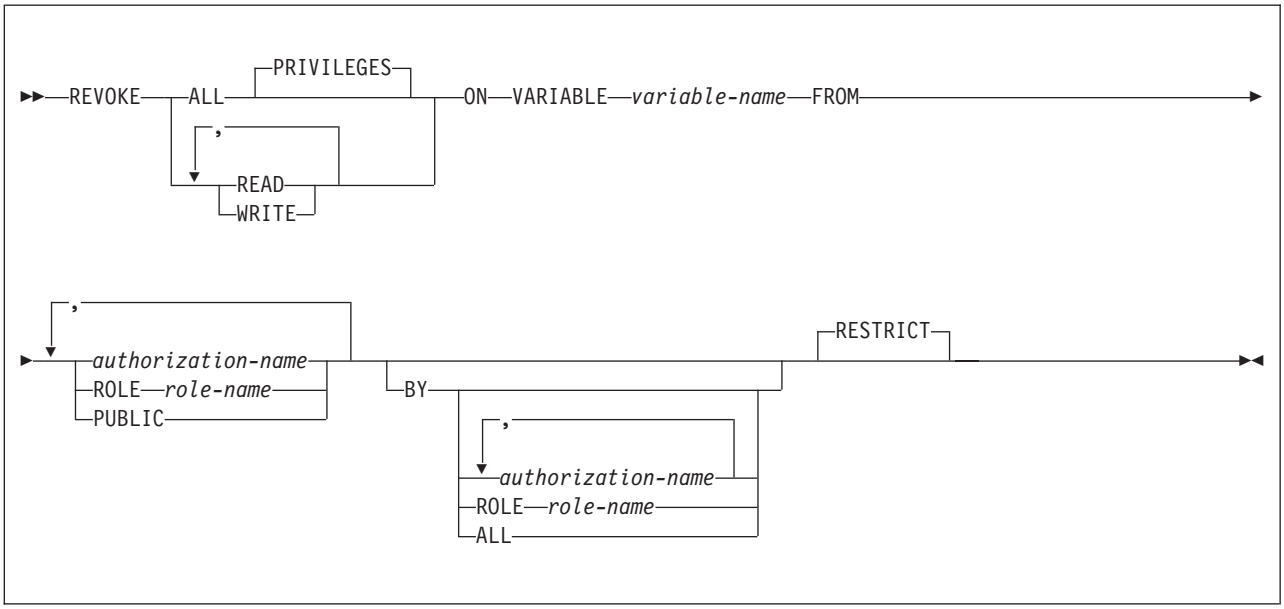
Example 4: Revoke the USAGE privilege on distinct type MILES from the role ROLE1:

```
REVOKE USAGE ON TYPE MILES
      FROM ROLE ROLE1;
```

REVOKE (variable privileges)

This form of the REVOKE statement revokes privileges on global variables.

Syntax



Description

ALL PRIVILEGES
Revokes both READ and WRITE privileges on the specified global variable.

READ
Revokes the privilege to read the value of the specified global variable.

WRITE
Revokes the privilege to assign a value to the specified global variable.

ON VARIABLE *variable-name*
Identifies the global variable from which privileges are revoked. *variable-name* must identify a global variable that exists at the current server.

FROM
Refer to "REVOKE" on page 1812 for a description of the FROM clause.

BY Refer to "REVOKE" on page 1812 for a description of the BY clause.

RESTRICT
Prevents the specified privileges from being revoked on a global variable if the following conditions exist:

- A function that is owned by the revokee references (READ or WRITE privilege) the specified global variable
- A view that is owned by the revokee references (READ or WRITE privilege) the specified global variable
- A trigger that is owned by the revokee references (READ or WRITE privilege) the specified global variable

Notes

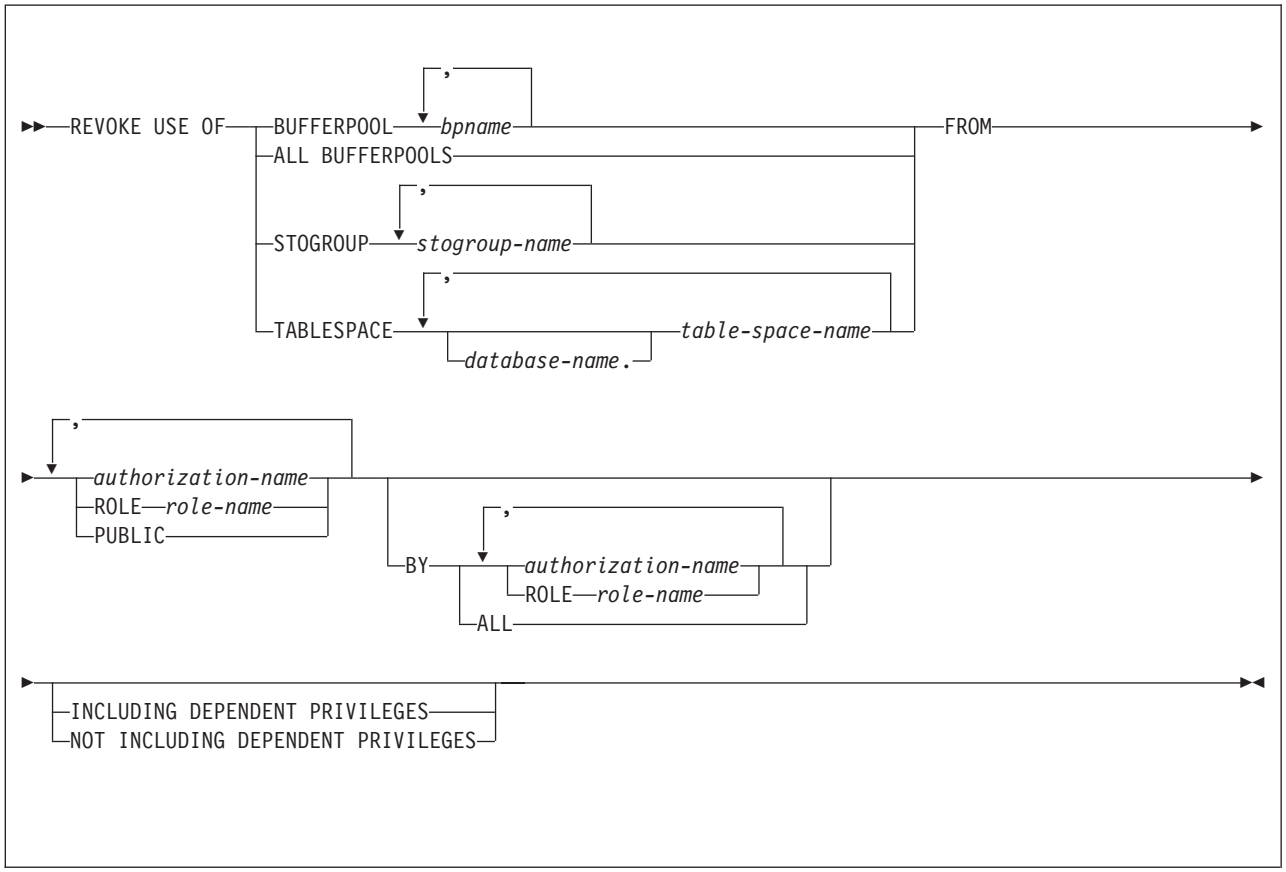
Global variables and statements in the dynamic statement cache: If a cached dynamic statement depends on the revoked authorization for the specified global variable and the cache statement is not in use, the cached dynamic statement will be invalidated.

Examples

REVOKE (use privileges)

This form of the REVOKE statement revokes authority to use particular buffer pools, storage groups, or table spaces.

Syntax



Description

BUFFERPOOL *bpname,...*
Revokes the privilege to refer to any of the identified buffer pools in a CREATE INDEX, CREATE TABLESPACE, ALTER INDEX, or ALTER TABLESPACE statement. See "Naming conventions" on page 57 for more details about *bpname*.

ALL BUFFERPOOLS
Revokes the privilege to refer to any buffer pool in a CREATE INDEX, CREATE TABLESPACE, ALTER INDEX, or ALTER TABLESPACE statement.

STOGROUP *stogroup-name,...*
Revokes the privilege to refer to any of the identified storage groups in a CREATE INDEX, CREATE TABLESPACE, ALTER INDEX, or ALTER TABLESPACE statement.

TABLESPACE *database-name.table-space-name,...*
Revokes the privilege to refer to any of the specified table spaces in a CREATE TABLE statement. The default *database-name* is DSNDB04.

For table spaces in a work file database you cannot revoke the privilege from PUBLIC. When a table space is created in a work file database, PUBLIC

implicitly receives the TABLESPACE privilege (without GRANT authority); this privilege is not recorded in the DB2 catalog, and it cannot be revoked.

FROM
Refer to "REVOKE" on page 1812 for a description of the FROM clause.

BY Refer to "REVOKE" on page 1812 for a description of the BY clause.

INCLUDING DEPENDENT PRIVILEGES or NOT INCLUDING DEPENDENT PRIVILEGES
Specifies whether revoking a privilege or an authority from an authorization ID or a role also results in revoking the grants that were made by that user. The default value is based on the authority that is being revoked and the REVOKE_DEP_PRIVILEGES system parameter:

- When ACCESSCTRL, DATAACCESS, or system DBADM authority is revoked, **NOT INCLUDING DEPENDENT PRIVILEGES** is assumed and the clause must be specified on the REVOKE statement.
- When the REVOKE_DEP_PRIVILEGES system parameter is set to NO, **NOT INCLUDING DEPENDENT PRIVILEGES** is assumed and an error is returned if the statement includes **INCLUDING DEPENDENT PRIVILEGES**.
- Otherwise, **INCLUDING DEPENDENT PRIVILEGES** is assumed and the clause must be specified on the REVOKE statement.

INCLUDING DEPENDENT PRIVILEGES
Specifies that revoking a privilege or an authority from an authorization ID or a role also results in revoking dependent privileges. This means that any grants that were made by the user will continue to be revoked, until all grants in the chain have been revoked.

INCLUDING DEPENDENT PRIVILEGES cannot be specified if the system parameter REVOKE_DEP_PRIVILEGES is set to NO, which enforces the behavior to not include the dependent privileges.

NOT INCLUDING DEPENDENT PRIVILEGES
Specifies that revoking a privilege or an authority from an authorization ID or a role does not cause the grants that were made by the user to be revoked. However, for the revoked privileges, all implications of the privilege being revoked are applied. For example, if the revoked privileges were required to bind a package successfully, that package would continue to be invalidated as a result of the package owner losing these privileges. An object might be dropped if a privilege is revoked that was used to create the object.

NOT INCLUDING DEPENDENT PRIVILEGES must be specified when ACCESSCTRL, DATAACCESS, or system DBADM authority is revoked.

NOT INCLUDING DEPENDENT PRIVILEGES cannot be specified if the system parameter REVOKE_DEP_PRIVILEGES is set toYES, which enforces the behavior to include dependent privileges in the revoke.

Notes

You can revoke privileges for only one type of object with each statement. Thus you can revoke the use of several table spaces with one statement, but not the use of a table space and a storage group.

For each object you name, you (or the indicated grantors) must have granted the USE privilege on that object to all identified users (including PUBLIC, if specified). The same object must not be identified more than once.

Revoking the privilege USE OF ALL BUFFERPOOLS does not cascade to all other privileges that can be granted under that privilege. A user with the privilege USE OF ALL BUFFERPOOLS WITH GRANT OPTION can make two types of grants:

- GRANT USE OF ALL BUFFERPOOLS TO *userid*. This privilege is revoked when the original user's privilege is revoked.
- GRANT USE OF BUFFERPOOL BP*n* TO *userid*. This privilege is *not revoked* when the original user's privilege is revoked.

Examples

Example 1: Revoke authority to use buffer pool BP2 from user MARINO.

```
REVOKE USE OF BUFFERPOOL BP2
  FROM MARINO;
```

Example 2: Revoke a grant of the USE privilege on the table space DSN8S11D in the database DSN8D11A. The grant is to PUBLIC, that is, to everyone at the local DB2 subsystem. (Grants to specific users are not affected.)

```
REVOKE USE OF TABLESPACE DSN8D11A.DSN8S11D
  FROM PUBLIC;
```

Example 3: Revoke the authority to use storage group SG1 from role ROLE1:

```
REVOKE USE OF STOGROUP SG1
  FROM ROLE ROLE1;
```

ROLLBACK

The ROLLBACK statement can be used to end a unit of recovery and back out all the relational database changes that were made by that unit of recovery. If relational databases are the only recoverable resources used by the application process, ROLLBACK also ends the unit of work. ROLLBACK can also be used to back out only the changes made after a savepoint was set within the unit of recovery without ending the unit of recovery. Rolling back to a savepoint enables selected changes to be undone.

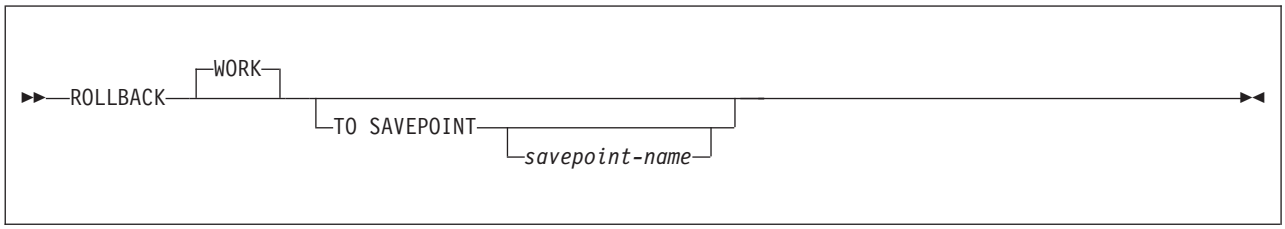
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared. It can be used in the IMS or CICS environment only if the TO SAVEPOINT clause is specified.

Authorization

None required.

Syntax



Description

When ROLLBACK is used without the SAVEPOINT clause, the unit of recovery in which the ROLLBACK statement is executed is ended and a new unit of recovery is started.

All changes that are made by the following statements during the unit of recovery are backed out:

- ALTER
- COMMENT
- CREATE
- DELETE
- DROP
- EXPLAIN
- GRANT
- INSERT
- LABEL
- MERGE
- REFRESH TABLE
- RENAME
- REVOKE
- SELECT INTO with an SQL data change statement

- select-statement with an SQL data change statement
- TRUNCATE when the IMMEDIATE clause is not specified
- UPDATE

ROLLBACK without the TO SAVEPOINT clause also causes the following actions to occur:

- All locks that are implicitly acquired during the unit of recovery are released. See "LOCK TABLE" on page 1757 for an explanation of the duration of explicitly acquired locks.
- All cursors are closed, all prepared statements are destroyed, and any cursors that are associated with the prepared statements are invalidated.
- All rows and all logical work files of every created temporary table of the application process are deleted. (All the rows of a declared temporary table are not implicitly deleted. As with base tables, any changes that are made to a declared temporary table during the unit of recovery are undone to restore the table to its state at the last commit point.)
- All LOB locators, including those that are held, are freed.

TO SAVEPOINT

Specifies that the unit of recovery is not to be ended and that only a partial rollback (to a savepoint) is to be performed. If a savepoint name is not specified, rollback is to the last active savepoint. For example, if in a unit of recovery, savepoints A, B, and C are set in that order and then C is released, ROLLBACK TO SAVEPOINT causes a rollback to savepoint B.

savepoint-name

Identifies the savepoint to which to roll back. The name must identify a savepoint that exists at the current server.

All database changes (including changes made to a declared temporary tables but excluding changes made to created temporary tables) that were made after the savepoint was set are backed out. Changes that are made to created temporary tables are not logged and are not backed out; a warning is issued instead. (A warning is also issued when a created temporary table is changed and there is an active savepoint.)

In addition, none of the following items are backed out:

- The opening or closing of cursors
- Changes in cursor positioning
- The acquisition and release of locks
- The caching of the rolled back statements

Any savepoints that are set after the one to which rollback is performed are released. The savepoint to which rollback is performed is not released.

ROLLBACK with or without the TO SAVEPOINT clause has no effect on connections.

Notes

The following information applies only to rolling back all changes in the unit of recovery (the ROLLBACK statement without the TO SAVEPOINT clause):

- *Stored procedures.* The ROLLBACK statement cannot be used if the procedure is in the calling chain of a user-defined function or a trigger or if DB2 is not the commit coordinator.

- *IMS or CICS.* Using a ROLLBACK to SAVEPOINT statement in an IMS or CICS environment only rolls back DB2 resources. Any other recoverable resources updated in the environment are not rolled back. To do a rollback operation in these environments, SQL programs must use the call prescribed by their transaction manager. The effect of these rollback operations on DB2 data is the same as that of the SQL ROLLBACK statement.

  A rollback operation in an IMS or CICS environment might handle the closing of cursors that were declared with the WITH hold option differently than the SQL ROLLBACK statement does. If an application requests a rollback operation from CICS or IMS, but no work has been performed in DB2 since the last commit point, the rollback request will not be broadcast to DB2. If the application had opened cursors using the WITH HOLD option in a previous unit of work, the cursors will not be closed, and any prepared statements associated with those cursors will not be destroyed.
- *Implicit rollback operations:* In all DB2 environments, the abend of a process is an implicit rollback operation.

ROLLBACK and non-LOB table spaces that are not logged: If ROLLBACK is executed for a unit of work that includes changes to a non-LOB table space that is not logged (specifies the NOT LOGGED attribute), that table space is marked RECOVER-pending and the table space is placed in the logical page list. The table space is therefore not available after the rollback operation completes. See *DB2 Utility Guide and Reference* for more information about the RECOVER utility.

ROLLBACK and declared global temporary tables that are not logged: When NOT LOGGED is specified on a declared global temporary table and DB2 must roll back because of an error such as a duplicate key error, rows are deleted or preserved depending on the option that was specified for ON ROLLBACK.

If the ON ROLLBACK DELETE ROWS option was specified for the table, insert, update, and delete activity is not logged. During a ROLLBACK or ROLLBACK TO SAVEPOINT operation, if the table was updated since the last COMMIT statement, all rows are deleted from the table. Any open cursors for the table do not have positions. If the declaration of the declared global temporary table was not committed, the declaration of the table is rolled back.

If the ON ROLLBACK PRESERVE ROWS option was specified for the table, insert, update, and delete activity is not logged. During a ROLLBACK or ROLLBACK TO SAVEPOINT operation, all rows in the table are preserved regardless of any updates to the table since the last COMMIT statement. Any open cursors for the table do not have positions. If the declaration of the declared global temporary table was not committed, the declaration of the table is rolled back.

Effect of rollback operations on global variables: Global variables are not controlled at the transaction level. Issuing a ROLLBACK statement does not effect the contents of a global variable.

Examples

Example 1: Roll back all DB2 database changes made since the unit of recovery was started.

```
ROLLBACK WORK;
```

Example 2: After a unit of recovery started, assume that three savepoints A, B, and C were set and that C was released:

```
...
SAVEPOINT A ON ROLLBACK RETAIN CURSORS;
...
SAVEPOINT B ON ROLLBACK RETAIN CURSORS;
...
SAVEPOINT C ON ROLLBACK RETAIN CURSORS;
...
RELEASE SAVEPOINT C;
...
```

Roll back all DB2 database changes only to savepoint A:

```
ROLLBACK WORK TO SAVEPOINT A;
```

If a savepoint name was not specified (that is, ROLLBACK WORK TO SAVEPOINT), the rollback would be to the last active savepoint that was set, which is B.

SAVEPOINT

The SAVEPOINT statement sets a savepoint within a unit of recovery to identify a point in time within the unit of recovery to which relational database changes can be rolled back.

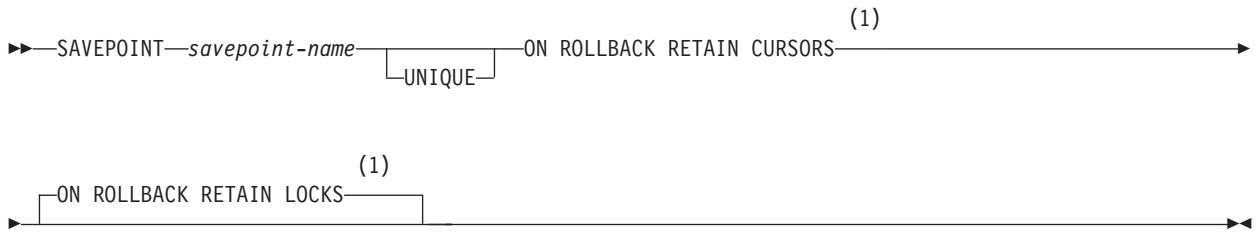
Invocation

This statement can be imbedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Notes:

1 These clauses can be specified in either order.

Description

savepoint-name
Names the savepoint. *savepoint-name* must not begin with 'SYS'.

UNIQUE
Specifies that the application program cannot reuse the savepoint name within the unit of recovery. An error occurs if a savepoint with the same name as *savepoint-name* already exists within the unit of recovery.

Omitting UNIQUE indicates that the application can reuse the savepoint name within the unit of recovery. If *svpt-name* identifies a savepoint that already exists within the unit of recovery and the savepoint was not created with the UNIQUE option, the existing savepoint is destroyed and a new savepoint is created. Destroying a savepoint to reuse its name for another savepoint is not the same as releasing the savepoint. Reusing a savepoint name destroys only one savepoint. Releasing a savepoint with the RELEASE SAVEPOINT statement releases the savepoint and all savepoints that have been subsequently set.

ON ROLLBACK RETAIN CURSORS
Specifies that any cursors that are opened after the savepoint is set are not tracked, and thus, are not closed upon rollback to the savepoint. Although these cursors remain open after rollback to the savepoint, they might not be usable. For example, if rolling back to the savepoint causes the insertion of a

row on which the cursor is positioned to be rolled back, using the cursor to update or delete the row results in an error.

ON ROLLBACK RETAIN LOCKS
Specifies that any locks that are acquired after the savepoint is set are not tracked, and thus, are not released on rollback to the savepoint. ON ROLLBACK RETAIN LOCKS is the default behavior.

Example

Assume that you want to set three savepoints at various points in a unit of recovery. Name the first savepoint A and allow the savepoint name to be reused. Name the second savepoint B and do not allow the name to be reused. Because you no longer need savepoint A when you are ready to set the third savepoint, reuse A as the name of the savepoint.

```
SAVEPOINT A ON ROLLBACK RETAIN CURSORS;

⋮

SAVEPOINT B UNIQUE ON ROLLBACK RETAIN CURSORS;

⋮

SAVEPOINT A ON ROLLBACK RETAIN CURSORS;
```

SELECT

The *select-statement* is the form of a query that can be directly specified in a DECLARE CURSOR statement, or prepared and then referenced in a DECLARE CURSOR statement. It can also be issued interactively using SPUFI or the command line processor which causes a result table to be displayed at your terminal. In any case, the table specified by *select-statement* is the result of the fullselect.

For a description of the SELECT statement, see “select-statement” on page 819.

SELECT INTO

The SELECT INTO statement produces a result table that contains at most one row. The statement assigns the values in that row to variables. If the table is empty, the statement does not assign values to the host variables or global variables.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

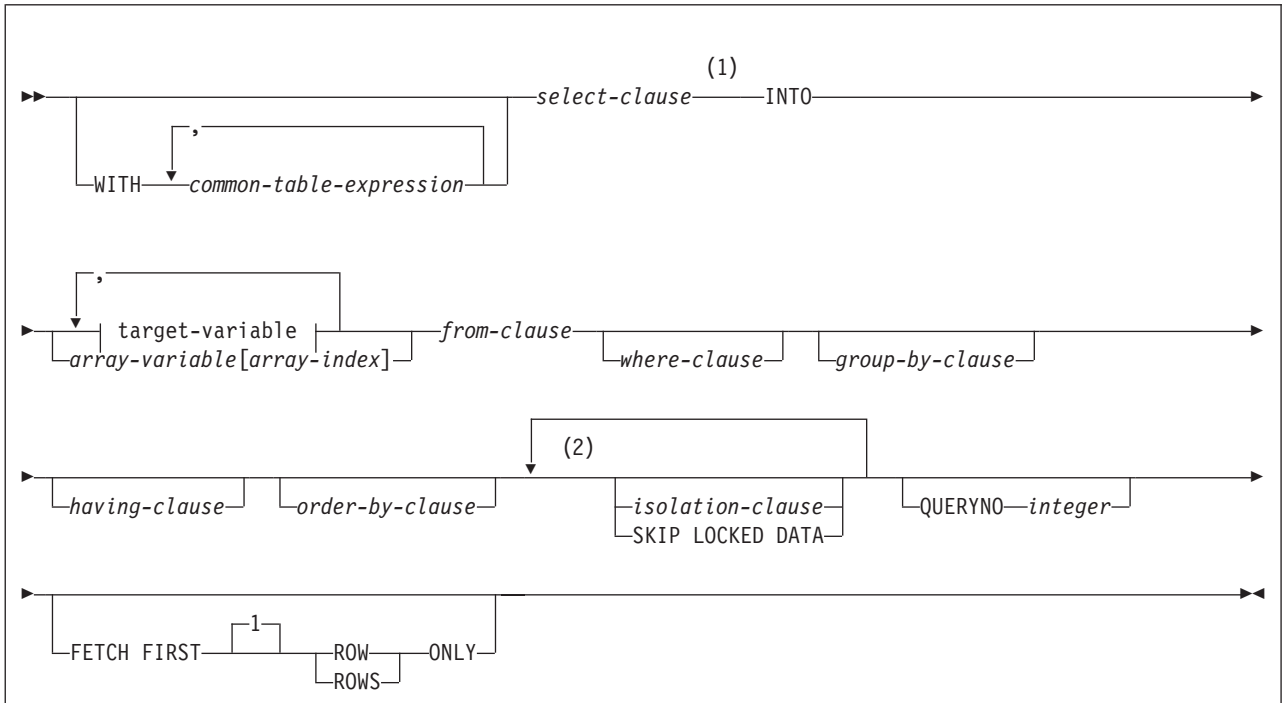
The privilege set that is defined below must include at least one of the following:

- The SELECT privilege on every table and view identified in the statement
- Ownership of every table and view identified in the statement
- READ and WRITE privileges on any global variables that are identified in the statement
- Ownership of any global variables that are identified in the statement
- DBADM authority for the database (tables only)
- DATAACCESS authority
- SYSADM authority
- SYSCTRL authority (catalog tables only)

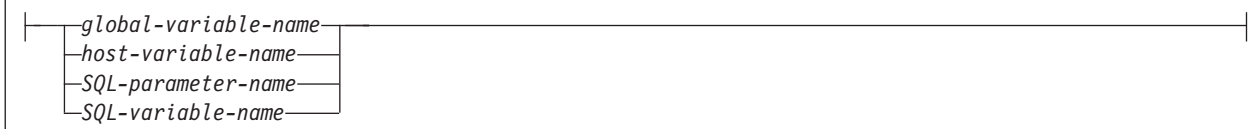
If the SELECT INTO statement includes an SQL data change statement, the privilege set must also include at least the privileges (INSERT, UPDATE, or DELETE) that are associated with that SQL data change statement on the table or view.

Privilege set: If the statement is embedded in an application program, the privilege set is the set of privileges that are held by the owner of the package.

Syntax



target-variable:



Notes:

1 The *select-clause* cannot reference both a system-period temporal table and an archive-enabled table.

2 The same clause must not be specified more than once.

Description

The table is derived by evaluating the *isolation-clause*, *from-clause*, *where-clause*, *group-by-clause*, *having-clause*, *order-by-clause*, and the *select-clause*, in this order. See Chapter 4, "Queries," on page 761 for a description of these clauses.

The tables or views identified in the statement can exist at the current server or at any DB2 subsystem with which the current server can establish a connection.

WITH *common-table-expression*
Refer to "common-table-expression" on page 820 for information about specifying a *common-table-expression*.

INTO *target-variable* **or** *array-variable*[*array-index*]
Identifies one or more targets for the assignment of output values. The number

of targets in the INTO clause must equal the number of values that are to be assigned. The first value in the result row is assigned to the first target in the list, the second value to the second target, and so on. A target variable must not be specified more than once in the INTO clause. Each assignment to a target is made in sequence through the list, according to the rules described in "Assignment and comparison" on page 121.

The value 'W' is assigned to the SQLWARN3 field of the SQLCA if the number of targets is less than the number of result column values.

If an error occurs on any assignment, the value is not assigned to the target, and no more values are assigned to the specified targets. Any values that have already been assigned remain assigned.

global-variable-name
Identifies the global variable that is the assignment target.

host-variable-name
Identifies the host variable that is the assignment target. For LOB output values, the target can be a regular host variable (if it is large enough), a LOB locator variable, or a LOB file reference variable.

SQL-parameter-name
Identifies the parameter that is the assignment target.

SQL-variable-name
Identifies the SQL variable that is the assignment target. SQL variables must be declared before they are used.

array-variable **[***array-index***]**
Specifies an array element that is the target of the assignment.

An array element must not be specified as the target for an assignment if *common-table-expression* is also specified in the statement.

[*array-index***]**
An expression that specifies which element in the array is the target of the assignment.

For an ordinary array, the array index expression must be castable to INTEGER, and must not be the null value. The index value must be between 1 and the maximum cardinality that is defined for the array.

For an associative array, the array index expression must be castable to the index data type of the associative array, and must not be the null value.

array-index must not be:

- An expression that references the CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP special register
- A nondeterministic function
- A function that is defined with EXTERNAL ACTION
- A function that is defined with MODIFIES SQL DATA
- A sequence expression

The data type of a variable must be compatible with the value assigned to it. If the value is numeric, the variable must have the capacity to represent the integral part of the value. For a date or time value, the variable must be a character string variable of a minimum length as defined in "Assignment and comparison" on page 121.

Each assignment to a variable is made according to the rules described in "Assignment and comparison" on page 121. Assignments are made in sequence through the list.

If an error occurs as the result of an arithmetic expression in the SELECT list of a SELECT INTO statement (division by zero or overflow) or a numeric conversion error occurs, the result is the null value. As in any other case of a null value, an indicator variable must be provided and the main variable is unchanged. In this case, however, the indicator variable is set to -2. Processing of the statement continues as if the error had not occurred. (However, this error causes a positive SQLCODE.) If you do not provide an indicator variable, a negative value is returned in the SQLCODE field of the SQLCA. Processing of the statement terminates when the error is encountered.

If an error occurs, no value is assigned to the variable or to later variables, though any values that have already been assigned to variables remain assigned.

If an error occurs because the result table has more than one row, values might be assigned to the variables. If values are assigned to the variables, the row that is the source of the values is undefined and not predictable.

isolation-clause
Specifies the isolation level at which the statement is executed and, optionally, the type of locks that are acquired.

SKIP LOCKED DATA
Specifies that rows are skipped when incompatible locks are held on the row by other transactions. These rows can belong to any accessed table that is specified in the statement. SKIP LOCKED DATA can be used only when isolation CS or RS is in effect and applies only to row level or page level locks.

SKIP LOCKED DATA is ignored if it is specified when the isolation level that is in effect is repeatable read (WITH RR) or uncommitted read (WITH UR).

QUERYNO *integer*
Specifies the number to be used for this SQL statement in EXPLAIN output and trace records. The number is used for the QUERYNO columns of the plan tables for the rows that contain information about this SQL statement. This number is also used in the QUERYNO column of the SYSIBM.SYSSTMT and SYSIBM.SYSPACKSTMT catalog tables.

If the clause is omitted, the number associated with the SQL statement is the statement number assigned during precompilation. Thus, if the application program is changed and then precompiled, that statement number might change.

Using the QUERYNO clause to assign unique numbers to the SQL statements in a program is helpful:
- For simplifying the use of optimization hints for access path selection
- For correlating SQL statement text with EXPLAIN output in the plan table

For information on using optimization hints, such as enabling the system for optimization hints and setting valid hint values, and for information on accessing the plan table, see *DB2 Performance Monitoring and Tuning Guide*.

FETCH FIRST ROW ONLY *integer*
The FETCH FIRST ROW ONLY clause can be used in the SELECT INTO statement when the query can result in more than a single row. The clause

indicates that only one row should be retrieved regardless of how many rows might be in the result table. When a number is explicitly specified, it must be 1.

Using the FETCH FIRST ROW ONLY clause to explicitly limit the result table to a single row provides a way for the SELECT INTO statement to be used with a query that returns more than a single row. Using the clause helps you to avoid using a cursor when you know that you want to retrieve only one row. To influence which row is returned, you can use the *order-by-clause*. When you specify *order-by-clause*, the rows of the result are ordered and then the first row is returned. If the FETCH FIRST ROW ONLY clause is not specified and the result table contains more than a single row, an error occurs.

Notes

Assignment to targets:

The *n*th target identified by the INTO clause or described in the SQLDA corresponds to the *n*th column of the result table of the cursor. The data type of target must be compatible with its corresponding value. If the value is numeric, the target must have the capacity to represent the whole part of the value. For a datetime value, the target must be a character string variable of a minimum length as defined in "String representations of datetime values" on page 101. When the target is a host variable, if the value is null, an indicator variable must be specified.

Assignments are made in sequence through the list. Each assignment to a target is made according to the rules described in Chapter 2, "Language elements," on page 53. If the number of targets is less than the number of values in the row, the SQLWARN3 field of the SQLCA is set to 'W'. There is no warning if there are more targets than the number of result columns. If the target is a host variable and the value is null, an indicator variable must be provided. If an assignment error occurs, the value is not assigned to the target and no more values are assigned to targets. Any values that have already been assigned to targets remain assigned.

If more than one assignment is included in the same assignment statement, all expressions are evaluated before the assignments are performed. For example, a reference to a variable in an expression always uses the value of the variable prior to any assignment in the assignment statement.

Normally, you use LOB locators to assign and retrieve data from LOB columns. However, because of compatibility rules, you can also use LOB locators to assign data to targets with other data types. For more information on using locators, see Saving storage when manipulating LOBs by using LOB locators (DB2 Application programming and SQL).

A timestamp without time zone value must not be assigned to a timestamp with time zone target.

Default encoding scheme:

The default encoding scheme for the data is the value in the bind option ENCODING, which is the option for application encoding. If this statement is used with functions such as LENGTH or SUBSTRING that are operating on LOB locators, and the LOB data that is specifies by the locator is in a different encoding scheme from the ENCODING bind option, LOB materialization and character conversion occur. To avoid LOB materialization and character conversion, select the LOB data from the SYSIBM.SYSDUMMYA, SYSIBM.SYSDUMMYE, or SYSIBM.SYSDUMMYU sample table.

If the result table is empty:

If the table is empty, the statement assigns +100 to SQLCODE, '02000' to SQLSTATE, and does not assign values to the host variables or global variables.

Number of rows inserted:

If the SELECT INTO statement of the cursor contains an SQL data change statement, the SELECT INTO operation sets SQLERRD(3) to the number of rows inserted.

Examples

Example 1: Put the maximum salary in DSN8B10.EMP into the host variable MAXSALRY.

```
EXEC SQL SELECT MAX(SALARY)
  INTO :MAXSALRY
  FROM DSN8B10.EMP;
```

Example 2: Put the row for employee 528671, from DSN8B10.EMP, into the host structure EMPREC.

```
EXEC SQL SELECT * INTO :EMPREC
  FROM DSN8B10.EMP
  WHERE EMPNO = '528671'
END-EXEC.
```

Example 3: Put the row for employee 528671, from DSN8B10.EMP, into the host structure EMPREC. Assume that the row will be updated later and should be locked when the query executes.

```
EXEC SQL SELECT * INTO :EMPREC
  FROM DSN8B10.EMP
  WHERE EMPNO = '528671'
  WITH RS USE AND KEEP EXCLUSIVE LOCKS
END-EXEC.
```

Example 4: Using a SELECT INTO statement, retrieve the value of INTCOL1 from table T1 into an element in array MYINTARRAY1, which is indexed by the value of the expression INTCOL2+MYINTVAR+1.

```
SELECT INTCOL1 INTO MYINTARRAY1[INTCOL2+MYINTVAR+1]
 FROM T1
 WHERE INTCOL1 = MYINTARRAY1[INTCOL2] ;
```

SET CONNECTION

The SET CONNECTION statement establishes the database server of the process by identifying one of its existing connections.

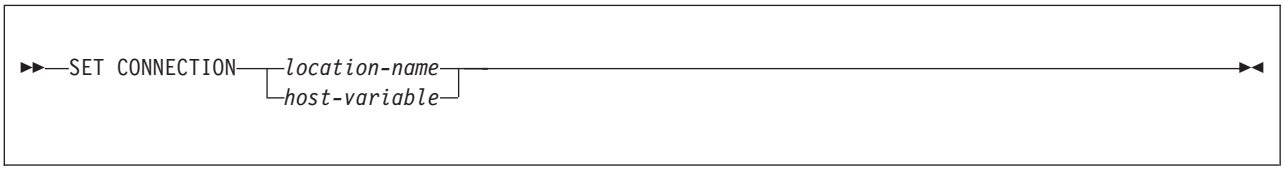
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

None required.

Syntax



Description

location-name **or** *host-variable*

Identifies the SQL connection by the specified location name or the location name contained in the host variable. If a host variable is specified:

- It must be a character string variable with a length attribute that is not greater than 16. (A C NUL-terminated character string can be up to 17 bytes.)
- It must not be followed by an indicator variable.
- The location name must be left-justified within the host variable and must conform to the rules for forming an ordinary location identifier.
- If the length of the location name is less than the length of the host variable, it must be padded on the right with blanks.

Let S denote the specified location name or the location name contained in the host variable. S must identify an existing SQL connection of the application process. If S identifies the current SQL connection, the state of S and all other connections of the application process are unchanged. The following rules apply when S identifies a dormant SQL connection.

If the SET CONNECTION statement is successful:

- SQL connection S is placed in the current state.
- S is placed in the CURRENT SERVER special register.
- Information about server S is placed in the SQLERRP field of the SQLCA. If the server is an IBM product, the information has the form *pppvvrrm*, where:
  - *ppp* is:
    ARI for DB2 Server for VSE & VM
    DSN for DB2 for z/OS
    QSQ for DB2 for i
    SQL for all other DB2 products
  - *vv* is a two-digit version identifier such as '11'.

– *rr* is a two-digit release identifier such as '01'.
– *m* is a one-digit modification level.
  - Values 0, 1, 2, 3, and 4 are reserved for modification levels in conversion and enabling-new-function mode from Version 10 (CM10, CM10*, ENFM10, and ENFM10*)
  - Values 5, 6, 7, 8, and 9 are for modification levels in new-function mode.

  For example, if the server is Version 9 of DB2 for z/OS in new-function mode with the latest maintenance, the value of SQLERRP is 'DSN09015'.

- Any previously current SQL connection is placed in the dormant state.

If the SET CONNECTION statement is unsuccessful, the connection state of the application process and the states of its SQL connections are unchanged.

Notes

SET CONNECTION after CONNECT (Type 1): The use of CONNECT (Type 1) statements does not prevent the use of SET CONNECTION, but the statement either fails or does nothing because dormant SQL connections do not exist. The SQLRULES(DB2) bind option does not prevent the use of SET CONNECTION, but the statement is unnecessary because CONNECT (Type 2) statements can be used instead. Use the SET CONNECTION statement to conform to the SQL standard.

Status of locks, cursors, and prepared statements: When an SQL connection is used, made dormant, and then restored to the current state in the same unit of work, the status of locks, cursors, and prepared statements for that SQL connection reflects its last use by the application process.

Host variables: If the SET CONNECTION statement contains host variables, the contents of the host variables are assumed to be in the encoding scheme that was specified in the ENCODING parameter when the package or plan that contains the statement was bound.

Restrictions on array types and array variables: In any SQL statement other than a CALL statement, array types and array variables must not be referenced after a connection at a remote server has been established. This restriction includes an SQL statement that executes at a remote server as a result of a three-part name or alias that resolves to an object at a remote server. An exception is that an array element can be the target of a FETCH, SELECT INTO, SET *assignment-statement*, or VALUES INTO statement in an SQL routine even when the statement is executed at a remote server.

Example

Execute SQL statements at TOROLAB1, execute SQL statements at TOROLAB2, and then execute more SQL statements at TOROLAB1.

```
EXEC SQL CONNECT TO TOROLAB1;

  -- execute statements referencing objects at TOROLAB1

EXEC SQL CONNECT TO TOROLAB2;

  -- execute statements referencing objects at TOROLAB2

EXEC SQL SET CONNECTION TOROLAB1;

  -- execute statements referencing objects at TOROLAB1
```

The first CONNECT statement creates the TOROLAB1 connection, the second CONNECT statement places it in the dormant state, and the SET CONNECTION statement returns it to the current state.

SET assignment-statement

The SET *assignment-statement* statement assigns values to variables and array elements.

Invocation

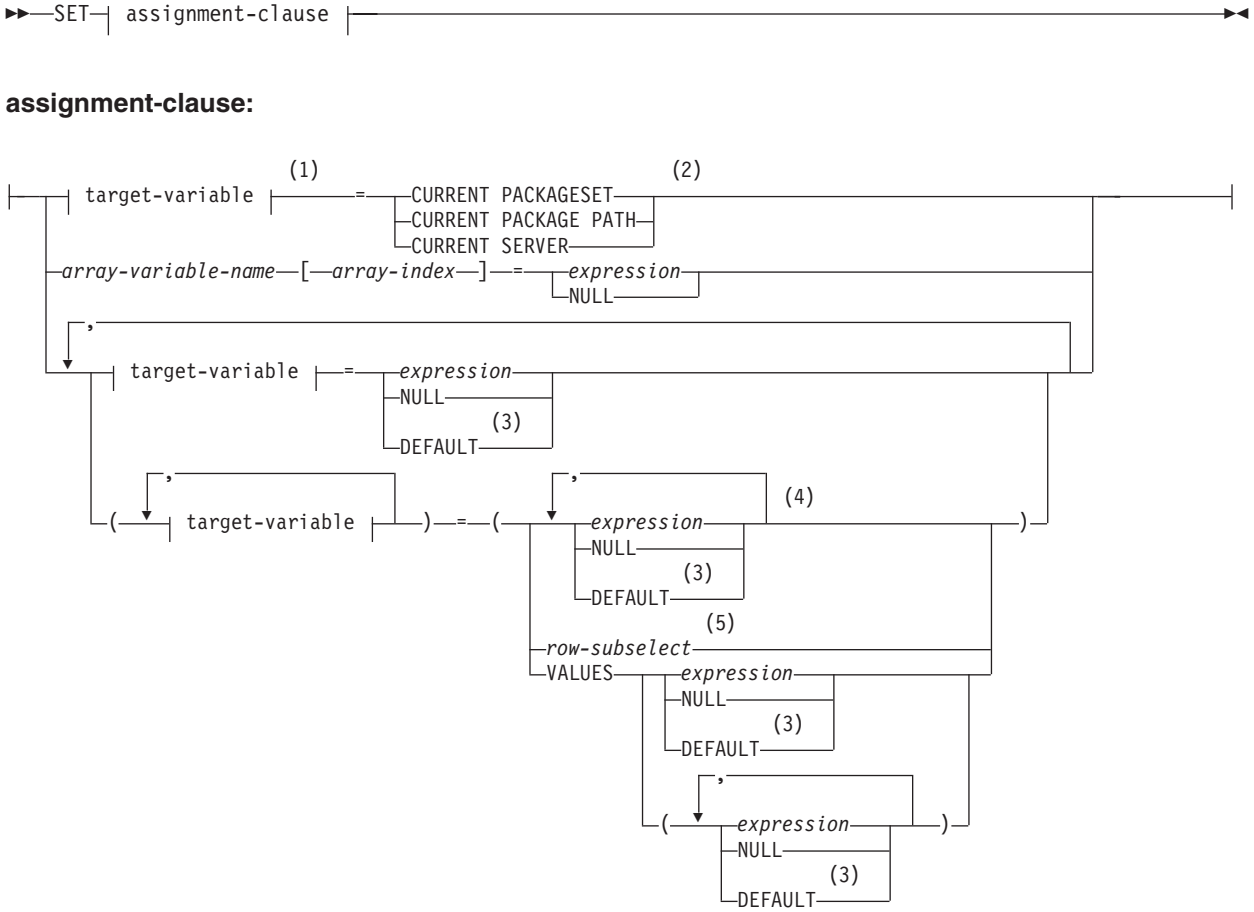
If all targets for the assignment statement are global variables, this statement can be embedded in an application program or issued interactively. It is an executable statement that can by dynamically prepared.

Otherwise, this statement can be embedded only in an application program. It is an executable statement that cannot by dynamically prepared.

Authorization

The privileges that are held by the current authorization ID must include those required to execute any of the expressions.

Syntax



target-variable:

Notes:

1 *target-variable* must not be an array type in this context.

2 When the target is not a transition variable, these special registers can be referenced only as a source value in this form of the syntax for this statement.

3 DEFAULT must only be specified when the corresponding target is a transition variable.

4 The number of source value specifications (expression, NULL, or DEFAULT) on the right side of the equal sign must match the number of target specifications on the left side of the statement.

5 *row-subselect* can be specified only in the outermost subselect within SQL PL. *target-variable* cannot be a *global-variable-name* or *transition-variable-name*.

Description

target-variable
Identifies one or more targets for the assignment of values. The number of targets must equal the number of values that are to be assigned.

The value that is to be assigned to each target variable can be specified immediately following the variable. For example:

```
variable=expression, variable=expression
```

Alternatively, sets of parentheses can be used to specify all of the target variables, and then all of the values. For example:

```
(variable,variable)=(expression,expression)
```

The value that is to be assigned to an array element must be specified immediately following the array element. For example:

```
array-variable[array-index]=expression
```

The data type of each variable in the variable list must be compatible with its corresponding result column. Each assignment to a *target-variable* is made in sequence through the list, according to the rules described in "Assignment and comparison" on page 121.

The value 'W' is assigned to the SQLWARN3 field of the SQLCA if the number of targets is less than the number of result column values.

If an error occurs on any assignment, the value is not assigned to the target, and no more values are assigned to the specified targets. Any values that have already been assigned remain assigned.

global-variable-name
Identifies the global variable that is the assignment target.

host-variable-name
Identifies the host variable that is the assignment target. For LOB output values, the target can be a regular host variable (if it is large enough), a LOB locator variable, or a LOB file reference variable.

SQL-parameter-name
Identifies the parameter that is the assignment target.

SQL-variable-name
Identifies the SQL variable that is the assignment target. SQL variables must be declared before they are used.

transition-variable-name
Identifies the column that is to be updated in the transition row. A transition variable name must identify a column in the subject table of a trigger, and is optionally qualified by a correlation name that identifies the new value.

transition-variable-name must not correspond to a begin column or end column of a BUSINESS_TIME period, and must not be specified if the statement contains a *period-clause*.

array-variable **[***array-index***]**
Specifies an array element that is the target of the assignment.

An array element must not be specified as the target for an assignment if *common-table-expression* is also specified in the statement.

[*array-index*]
An expression that specifies which element in the array is the target of the assignment.

For an ordinary array, the array index expression must be castable to INTEGER, and must not be the null value. The index value must be between 1 and the maximum cardinality that is defined for the array.

For an associative array, the array index expression must be castable to the index data type of the associative array, and must not be the null value.

array-index must not be:

- An expression that references the CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP special register
- A nondeterministic function
- A function that is defined with EXTERNAL ACTION
- A function that is defined with MODIFIES SQL DATA
- A sequence expression

expression
Specifies the value that is to be assigned to the corresponding assignment target. The expression is any expression of the type described in "Expressions" on page 240, except that it cannot contain a reference to the CURRENT PACKAGESET, CURRENT PACKAGE PATH, or CURRENT SERVER special register. All expressions are evaluated before any result is assigned to a target. If an expression refers to a variable or array element that is used in the list of assignment targets, the value of the variable or array element in the expression is the value of the variable or array element prior to any assignments.

Each assignment to a target is made according to the assignment rules described in "Assignment and comparison" on page 121. When the target variables and expressions are in the following form, the first value is assigned to the first target variable in the list, the second value is assigned to the second target variable in the list, and so on.

```
(target-variable,target-variable,...)=(expression,expression,...)
```

NULL Specifies the null value and can only be specified for host variables that have an associated indicator variable.

VALUES
Specifies the values that are to be assigned to the corresponding assignment targets. When more than one value is specified, the values must be enclosed in parentheses. Each value can be an expression or NULL, as previously described. The following syntaxes are equivalent:

- (*target-variable*, *target-variable*) = (VALUES(*expression*, NULL))
- (*target-variable*, *target-variable*) = (*expression*, NULL)

The CURRENT APPLICATION ENCODING SCHEME, CURRENT PACKAGESET, CURRENT PACKAGE PATH, or CURRENT SERVER special registers can be referenced only in

a SET *assignment-statement* statement or SQL PL *assignment-statement* statement that results in the assignment of a single target.

A parameter marker must not be specified.

row-subselect
A subselect that returns a single row. The number of columns corresponds to the number of target variables that are specified for assignment. Each result column value is assigned to the corresponding variable. If the result of the row subselect is no rows, then null values are assigned. An error is returned if there is more than one row in the result. *row-subselect* can be specified only in the outermost subselect within SQL PL. The target variable must not be a global variable or transition variable.

Notes

Assignment to targets:
If more than one assignment is included in the same assignment statement, all expressions are evaluated before the assignments are performed. For example, a reference to a variable in an expression always uses the value of the variable prior to any assignment in the assignment statement.

Normally, you use LOB locators to assign and retrieve data from LOB columns. However, because of compatibility rules, you can also use LOB locators to assign data to targets with other data types. For more information on using locators, see Saving storage when manipulating LOBs by using LOB locators (DB2 Application programming and SQL).

Default encoding scheme:
The default encoding scheme for the data is the value in the bind option ENCODING, which is the option for application encoding. If this statement is used with functions such as LENGTH or SUBSTRING that are operating on LOB locators, and the LOB data that is specifies by the locator is in a different encoding scheme from the ENCODING bind option, LOB materialization and character conversion occur. To avoid LOB materialization and character conversion, select the LOB data from the SYSIBM.SYSDUMMYA, SYSIBM.SYSDUMMYE, or SYSIBM.SYSDUMMYU sample table.

Examples

Example 1: Set the host variable HVL to the value of the CURRENT PATH special register.

```
SET :HVL = CURRENT PATH;
```

Example 2: Set the host variable PATH to the contents of the SQL PATH special register, the host variable XTIME to the local time at the current server, and the host variable MEM to the current member of the data sharing environment.

```
SET :SERVER = CURRENT PATH,
    :XTIME = CURRENT TIME,
    :MEM = CURRENT MEMBER;
```

Example 3: Set the host variable DETAILS to a portion of a LOB value, using a LOB expression with a LOB locator to refer the extracted portion of the value.

```
SET :DETAILS = SUBSTR(:LOCATOR,1,35);
```

If the LOB data that is specified by the LOB locator LOCATOR is in a different encoding scheme from the value of the ENCODING bind option, and you want to avoid LOB materialization and character conversion, use the following statement instead of the SET statement:

```
SELECT SUBSTR(:LOCATOR,1,35)
   INTO :DETAILS
   FROM SYSIBM.SYSDUMMYU;
```

Example 4: Set host variable HV1 to the results of external function CALC_SALARY and host variable HV2 to the value of special register CURRENT PATH. Use an indicator value with HV1 in case CALC_SALARY returns a null value.

```
SET (:HV1:IND1, :HV2) =
    (CALC_SALARY(:HV3, :HF4), CURRENT PATH);
```

Example 5: Assume that you want to create a before trigger that sets the salary and commission columns to default values for newly inserted rows in the EMPLOYEE table and that you will define the trigger only with NEW in the REFERENCING clause. Assign the default values to the SALARY and COMMISSION columns.

```
SET (SALARY, COMMISSION) = (50000, 8000);
```

Example 6: Assume that you want to create a before trigger that detects any commission increases greater than 10% for updated rows in the EMPLOYEE table and limits the commission increase to 10%. You will define the trigger with both OLD and NEW in the REFERENCING clause. Limit an increase to the COMMISSION column to 10%.

```
SET NEWROW.COMMISSION = 1.1 * OLDROW.COMMISSION;
```

Example 7: Suppose that the associative array variable CANADACAPITALS has array type CAPITALSARRAY. Use SET *assignment-statement* statements to assign values to CANADACAPITALS.

```
SET CANADACAPITALS['British Columbia'] = 'Victoria';
SET CANADACAPITALS['Alberta'] = 'Edmonton';
SET CANADACAPITALS['Manitoba'] = 'Winnipeg';
SET CANADACAPITALS['Ontario'] = 'Toronto';
SET CANADACAPITALS['Nova Scotia'] = 'Halifax';
```

In the CANADACAPITALS array, the array index values are province names, and the associated array element values are the names of the corresponding capital cities. The order in which values are assigned to associative array elements does not matter. The elements of an associative array are stored in the array in ascending order of the associated array index values.

Example 8: Suppose that the associative array variables CANADACAPITALSA and CANADACAPITALSB have array type CAPITALSARRAY. The following SET *assignment-statement* statements have been used to assign values to CANADACAPITALSA.

```
SET CANADACAPITALSA['British Columbia'] = 'Victoria';
SET CANADACAPITALSA['Alberta'] = 'Edmonton';
SET CANADACAPITALSA['Manitoba'] = 'Winnipeg';
SET CANADACAPITALSA['Ontario'] = 'Toronto';
SET CANADACAPITALSA['Nova Scotia'] = 'Halifax';
```

Use a single SET *assignment-statement* statement to assign all of the values that are in CANADACAPITALSA to CANADACAPITALSB.

```
SET CANADACAPITALSB = CANADACAPITALSA;
```

Example 9: Suppose that P_PHONENUMBERS SQL array variable is defined as an ordinary array. Set P_PHONENUMBERS to an array of fixed numbers.

```
SET P_PHONENUMBERS = ARRAY[9055553907, 4165554213, 4085553678];
```

Example 10: Set the SQL array variable P_PHONENUMBERS to an array of numbers that are retrieved from the PHONENUMBER table.

```
SET P_PHONENUMBERS =
 ARRAY [SELECT NUMBER
 FROM PHONENUMBERS
 WHERE EMPID = 624];
```

Example 11: Suppose that no values have been assigned to SQL array variable P_PHONENUMBERS. Assign the value of SQL variable P_MYNUMBER to the first and tenth elements of P_PHONENUMBERS. After the first assignment, the cardinality of P_PHONENUMBERS is 1. After the second assignment, the cardinality is 10, and elements 2 to 9 have been implicitly assigned the null value.

```
SET P_PHONENUMBERS[1]  = P_MYNUMBER;
SET P_PHONENUMBERS[10] = P_MYNUMBER;
```

SET CURRENT APPLICATION COMPATIBILITY

The SET CURRENT APPLICATION COMPATIBILITY statement assigns a value to the CURRENT APPLICATION COMPATIBILITY special register. This special register allows users to control the package compatibility level behavior for dynamic SQL.

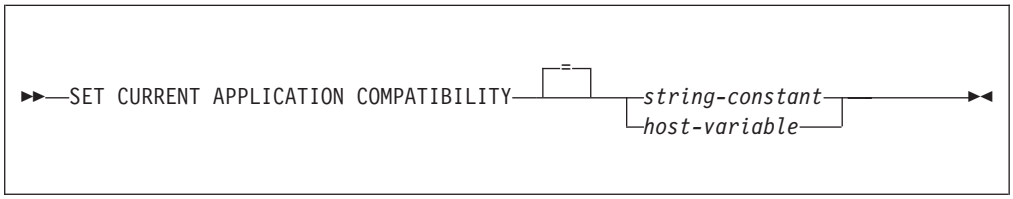
Invocation

This statement can be embedded only in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

None required.

Syntax



Description

string-constant
The following values are supported:

V10R1 The dynamic SQL statements in the package have V10R1 compatibility behavior.

V11R1 The dynamic SQL statements in the package will V11R1 compatibility behavior. This value is only allowed in new-function mode.

host-variable
A variable with a data type of CHAR or VARCHAR. The value of *host-variable* must not be null and must represent a valid release compatibility level.

The value must:
- Be left-aligned within the *host-variable*
- Be padded on the right with blanks if its length is less than the *host-variable*

Examples

The following examples set the CURRENT APPLICATION COMPATIBILITY special register to 'V10R1' (in the second example, Host variable HV1 = 'V10R1').

```
EXEC SQL SET CURRENT APPLICATION COMPATIBILITY = 'V10R1';
EXEC SQL SET CURRENT APPLICATION COMPATIBILITY = :HV1;
```

Related reference:

"CURRENT APPLICATION COMPATIBILITY" on page 161

SET CURRENT APPLICATION ENCODING SCHEME

The SET CURRENT APPLICATION ENCODING SCHEME statement assigns a value to the CURRENT APPLICATION ENCODING SCHEME special register. This special register allows users to control which encoding scheme will be used for dynamic SQL statements after the SET statement has been executed.

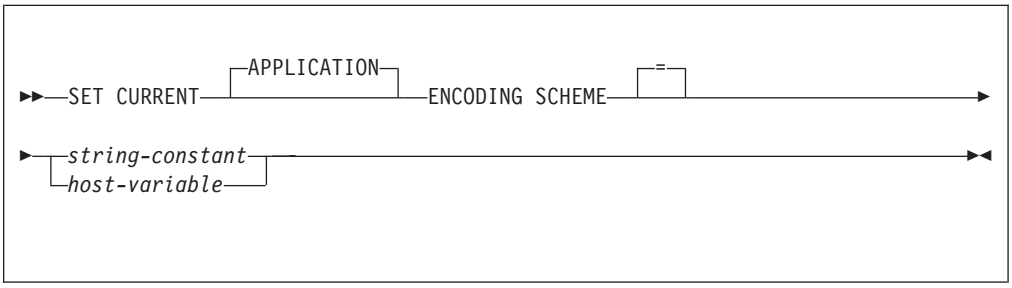
Invocation

This statement can be embedded only in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

None required.

Syntax



Description

string-constant
A character string constant that represents a valid encoding scheme (ASCII, EBCDIC, UNICODE, or a character representation of a number between 1 and 65533).

host variable
A variable with a data type of CHAR or VARCHAR. The value of *host-variable* must not be null and must represent a valid encoding scheme or a character representation of a number between 1 and 65533). An associated indicator variable must not be provided.

The value must:
- Be left justified within the host variable
- Be padded on the right with blanks if its length is less than that of the host variable

Examples

The following examples set the CURRENT APPLICATION ENCODING SCHEME special register to 'EBCDIC' (in the second example, Host variable HV1 = 'EBCDIC').

```
EXEC SQL SET CURRENT APPLICATION ENCODING SCHEME = 'EBCDIC';
EXEC SQL SET CURRENT ENCODING SCHEME  = :HV1;
```

Related reference:

"CURRENT APPLICATION ENCODING SCHEME" on page 162

SET CURRENT DEBUG MODE

The SET CURRENT DEBUG MODE statement assigns a value to the CURRENT DEBUG MODE special register.

The special register sets the default value for the DEBUG MODE option for the following statements:

- CREATE FUNCTION statements that define an SQL scalar function
- ALTER FUNCTION statements that create or replace a version of an SQL scalar function
- CREATE PROCEDURE statements that define a native SQL or Java procedure
- ALTER PROCEDURE statements that create or replace a version of a native SQL procedure

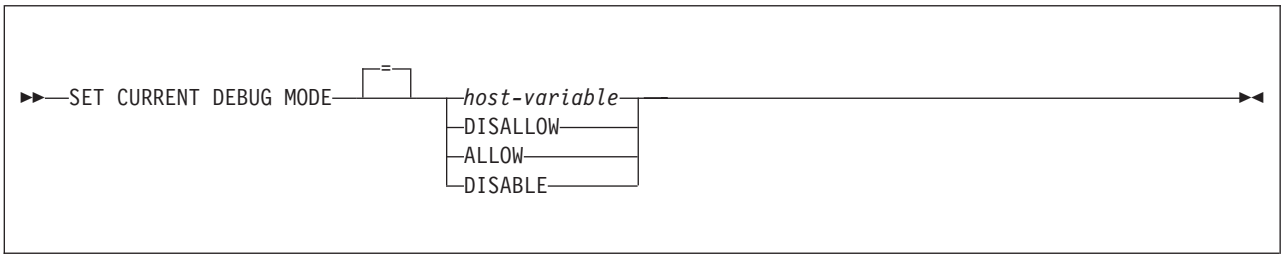
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

host-variable
Specifies a host variable that contains the debugging option. The host variable must conform to the following rules:

- Be a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC variable. The actual length of the contents of the host variable must not exceed the length of the special register.
- Include a keyword value of DISALLOW, ALLOW, or DISABLE that is left justified
- Be padded on the right with blanks if the host variable is a fixed length character
- Not contain lowercase letters or characters that cannot be specified in an ordinary identifier
- Not be empty or contain only blanks
- Not be the null value

DISALLOW
Specifies that DISALLOW DEBUG MODE is the default option for CREATE statements when defining an SQL scalar function, a native SQL procedure, or a

Java procedure, or ALTER statements that create or replace a version of an SQL scalar function or a native SQL procedure.

ALLOW
Specifies that ALLOW DEBUG MODE is the default option for CREATE statements when defining an SQL scalar function, a native SQL procedure, a Java procedure, or ALTER statements that create or replace a version of an SQL scalar function or a native SQL procedure.

DISABLE
Specifies that DISABLE DEBUG MODE is the default option for CREATE statements when defining an SQL scalar function, a native SQL procedure, a Java procedure, or ALTER statements that create or replace a version of an SQL scalar function or a native SQL procedure.

Examples

Example: The following statement sets the CURRENT DEBUG MODE special register so that the default option for CREATE PROCEDURE statements will be ALLOW DEBUG MODE:

```
SET CURRENT DEBUG MODE = ALLOW;
```

Related reference:

"CURRENT DEBUG MODE" on page 171

SET CURRENT DECFLOAT ROUNDING MODE

The SET CURRENT DECFLOAT ROUNDING MODE statement assigns a value to the CURRENT DECFLOAT ROUNDING MODE special register. The special register sets the default rounding mode that is used with decimal floating point values (DECFLOAT).

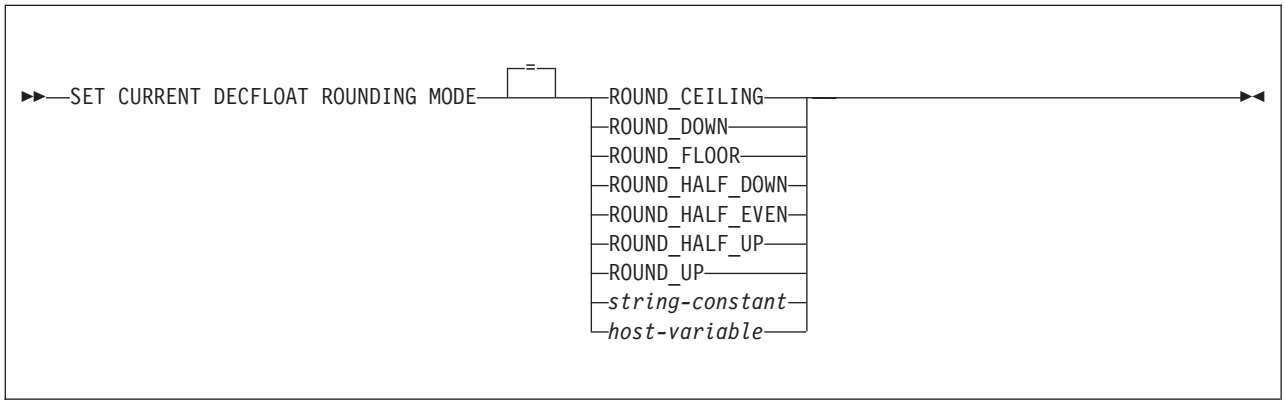
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

ROUND_CEILING
Round towards positive infinity. If all of the discarded digits are zero or if the sign is negative, the result is unchanged other than the removal of discarded digits. Otherwise, the result coefficient is incremented by 1 (round up).

ROUND_DOWN
Round towards 0 (truncation). The discarded digits are ignored.

ROUND_FLOOR
Round towards negative infinity. If all of the discarded digits are zero or if the sign is positive, the result is unchanged other than the removal of discarded digits. Otherwise, the sign is negative and the result coefficient is incremented by 1 (round down).

ROUND_HALF_DOWN
Round to nearest value; if values are equidistant, rounds down. If the discarded digits represent greater than half (0.5) of the value of a number in the next left position, the result coefficient is incremented by 1 (round up). Otherwise, the discarded digits are ignored. This rounding mode is not recommended when creating a portable application because it is not supported by the IEEE draft standard for floating-point arithmetic.

ROUND_HALF_EVEN
Round to nearest value; if values are equidistant, round so that the final digit

is even. If the discarded digits represent greater than half (0.5) of the value of a number in the next left position, the result coefficient is incremented by 1 (round up). If the discarded digits represent less than half of the value, the result coefficient is not adjusted (that is, the discarded digits are ignored). Otherwise, the result coefficient is unaltered if its rightmost digit is even, or is incremented by 1 (round up) if its rightmost digit is odd (to make an even digit).

ROUND_HALF_UP
Round to nearest value; if values are equidistant, round up. If the discarded digits represent greater than or equal to half (0.5) of the value of a number in the next left position, the result coefficient is incremented by 1 (round up). Otherwise the discarded digits are ignored.

ROUND_UP
Round away from 0. If all of the discarded digits are zero, the result is unchanged other than the removal of discarded digits. Otherwise, the result coefficient is incremented by 1 (round up). This rounding mode is not recommended when creating a portable application because it is not supported by the IEEE draft standard for floating-point arithmetic.

string-constant
Specifies a string constant that contains a specification of the rounding mode. The string-constant must have the following characteristics:

- Must be a string constant. The actual length of the contents of the string constant, after trailing blanks have been removed, must not exceed 19 characters.
- Must not be the null value.
- Must not contain lower case letters or characters that cannot be specified in an ordinary identifier.
- Must specify one of the seven rounding mode keywords as a string constant.

host-variable
Specifies a variable that contains a specification of the rounding mode. The variable must have the following characteristics:

- Must have a length, after trailing blanks have been removed, that does not exceed 19 bytes.
- Must not be followed by an indicator variable.
- Must not be a CLOB or DBCLOB.
- Must include a rounding mode that is left justified and conforms to the rules for forming an ordinary identifier.
- Must not contain lower case letters or characters that cannot be specified in an ordinary identifier.
- Must be padded on the right with blanks if the variable is a fixed length string.
- Must contain one of the seven rounding mode keywords.

Examples

Example: The following statement sets the CURRENT DECFLOAT ROUNDING MODE to ROUND_CEILING, using a string constant and a keyword.

```
SET CURRENT DECFLOAT ROUNDING MODE = ROUND_CEILING;
```

Related reference:
"CURRENT DECFLOAT ROUNDING MODE" on page 172

SET CURRENT DEGREE

The SET CURRENT DEGREE statement assigns a value to the CURRENT DEGREE special register.

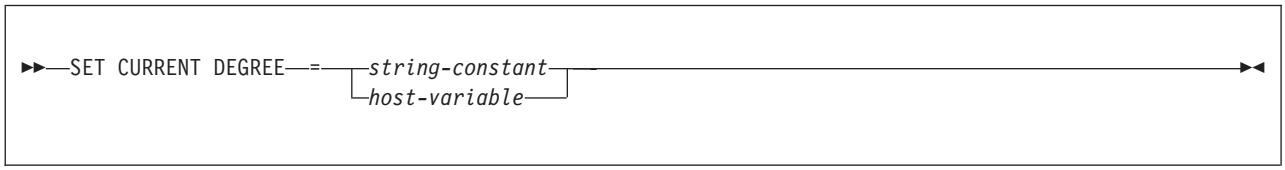
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

The value of CURRENT DEGREE is replaced by the value of the string constant or host variable. The value must be a character string that is not longer than 3 bytes and the value must be 'ANY', '1', or '1 '.

Notes

If the value of CURRENT DEGREE is '1' when a query is dynamically prepared, the execution of that query will not use parallel operations. If the value of CURRENT DEGREE is 'ANY' when a query is dynamically prepared, the execution of that query can involve parallel operations.

For distributed applications, the default value at the server is used unless the requesting application issues the SQL statement SET CURRENT DEGREE. For requests using DRDA, the SET CURRENT DEGREE statement must be within the scope of the CONNECT statement.

The value specified in the SET CURRENT DEGREE statement remains in effect until it is changed by the execution of another SET CURRENT DEGREE statement or until deallocation of the application process. For applications that connect to DB2 using the call attachment facility, the value of register CURRENT DEGREE can be requested to remain in effect for a longer duration. For more information, see the description of the call attachment facility CONNECT statement in *DB2 Application Programming and SQL Guide*.

Examples

Example 1: The following statement inhibits parallel operations:

```
SET CURRENT DEGREE = '1';
```

Example 2: The following statement allows parallel operations:

```
SET CURRENT DEGREE = 'ANY';
```

Related concepts:

Parallel processing (DB2 Performance)

Call attachment facility (DB2 Application programming and SQL)

Related tasks:

Enabling parallel processing (DB2 Performance)

Disabling query parallelism (DB2 Performance)

Related reference:

"CURRENT DEGREE" on page 174

CURRENT DEGREE field (CDSSRDEF subsystem parameter) (DB2 Installation and Migration)

"CONNECT" on page 1147

SET CURRENT EXPLAIN MODE

The SET CURRENT EXPLAIN MODE statement assigns a value to the CURRENT EXPLAIN MODE special register.

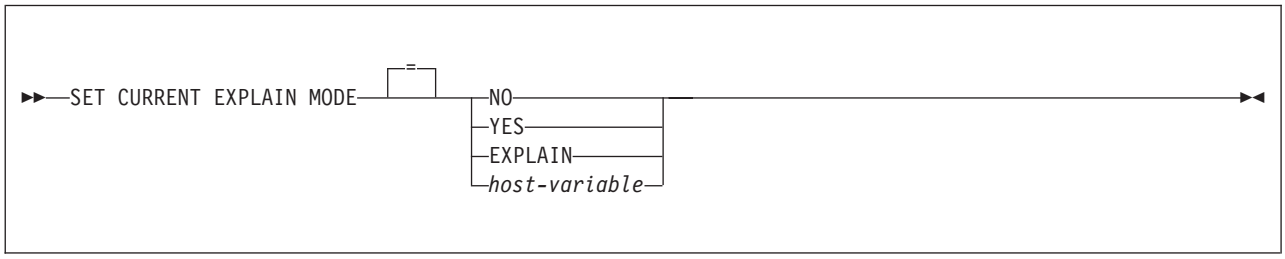
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

This statement replaces the value of the CURRENT EXPLAIN MODE special register with the value of the specified keyword or host variable.

NO Specifies that no EXPLAIN information is captured. NO is the initial value of the EXPLAIN MODE special register.

YES
Enables the EXPLAIN facility and causes EXPLAIN information to be inserted into the EXPLAIN tables for eligible dynamic SQL statements after the statement is prepared and executed. All dynamic SQL statements are compiled and executed normally.

EXPLAIN
Enables the EXPLAIN facility and causes EXPLAIN information to be captured for any eligible dynamic SQL statement after the statement is prepared. This setting behaves similarly to YES, however, dynamic statements, except for SET statements, are not executed.

host-variable
host-variable must be a CHAR or VARCHAR value and must be NO, YES, or EXPLAIN. Leading blanks are not allowed. All input values must be uppercase, must be left justified within the host variable, and must be padded on the right with blanks if the length of the value is less than the length of the host variable.

For values YES and EXPLAIN, prepared statements are not saved into the dynamic statement cache.

Examples

Example 1: The following statement sets the CURRENT EXPLAIN MODE special register, so that EXPLAIN information will be captured for any subsequent eligible dynamic SQL statements during execution.

```
SET CURRENT EXPLAIN MODE = YES;
```

Related reference:

"CURRENT EXPLAIN MODE" on page 175

SET CURRENT GET_ACCEL_ARCHIVE

The SET CURRENT GET_ACCEL_ARCHIVE statement changes the value of the CURRENT GET_ACCEL_ARCHIVE special register.

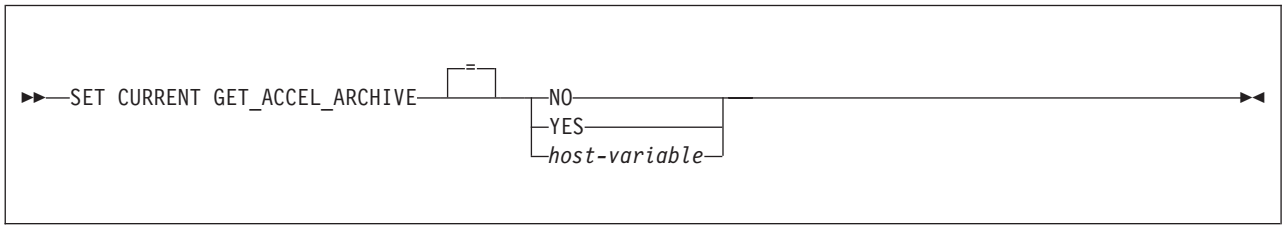
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

NO Specifies that if a table is archived in an accelerator server, and a query references that table, the query does not use the data that is archived.

YES
Specifies that if a table is archived in an accelerator server, and a query references that table, the query uses the data that is archived.

host-variable
A variable with a data type of CHAR or VARCHAR. The length must not exceed 255 bytes. Valid values are YES or NO. If *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value. The value of *host-variable* must be left justified and must be padded on the right with blanks.

Examples

The following statement sets the CURRENT GET_ACCEL_ARCHIVE special register to NO to indicate that when a table is archived in an accelerator server, the table reference does not include the archived data.

```
SET CURRENT GET_ACCEL_ARCHIVE=NO;
```

Related reference:

"CURRENT GET_ACCEL_ARCHIVE" on page 176

SET CURRENT LOCALE LC_CTYPE

The SET CURRENT LOCALE LC_CTYPE statement assigns a value to the CURRENT LOCALE LC_CTYPE special register. The special register allows control over the LC_CTYPE locale for statements that use a built-in function that refers to a locale, such as LCASE, UCASE, and TRANSLATE (with a single argument).

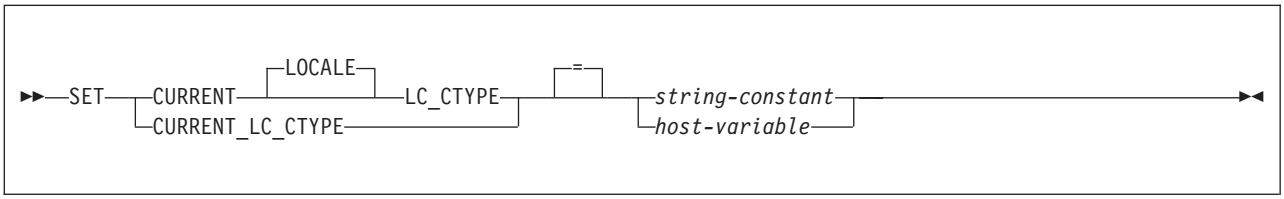
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

The value of CURRENT LOCALE LC_CTYPE is replaced by the value specified.

string-constant
A character string constant that must not be longer than 50 bytes and must represent a valid locale.

host-variable
A variable with a data type of CHAR or VARCHAR and a length that is not longer than 50 bytes. The value of *host-variable* must not be null and must represent a valid locale. If the host variable has an associated indicator variable, the value of the indicator variable must not indicate a null value.

The locale must:

- Be left justified within the host variable
- Be padded on the right with blanks if its length is less than that of the host variable

A locale can be specified in uppercase characters, lowercase characters, or a combination of the two. For more information, see CURRENT LOCALE LC_CTYPE.

Note: The existence of a locale is not validated when the CURRENT LOCALE LC_CTYPE special register is set. For example, a locale name that is misspelled is not detected, which could affect the way subsequent SQL operates. When the special register value is used at execution time, an error is returned if the locale does not exist. For example, if the LOWER function is invoked without specifying a locale name, the special register determines the locale that is used.

Examples

Example 1: Set the CURRENT LOCALE LC_CTYPE special register to the locale 'En_US'.

```
EXEC SQL SET CURRENT LOCALE LC_CTYPE = 'En_US';
```

Example 2: Set the CURRENT LOCALE LC_CTYPE special register to the value of host variable HV1, which contains 'Fr_FR@EURO'.

```
EXEC SQL SET CURRENT LOCALE LC_CTYPE = :HV1;
```

Related concepts:

z/OS: Unicode Services User's Guide and Reference

Related reference:

"CURRENT LOCALE LC_CTYPE" on page 177

SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION

The SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION statement changes the value of the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register.

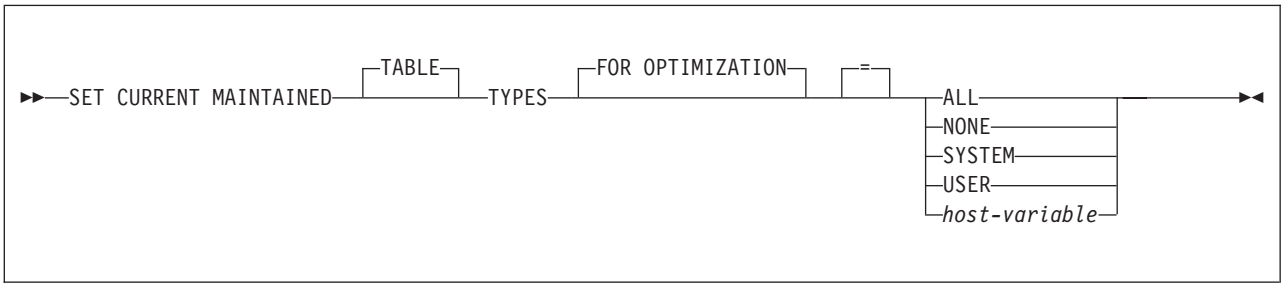
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

The value indicates which materialized query tables that are enabled for optimization are considered when optimizing the processing of dynamic SQL queries.

ALL
Indicates that all materialized query tables will be considered.

NONE
Indicates that no materialized query tables will be considered.

SYSTEM
Indicates that only system-maintained materialized query tables that are refresh deferred will be considered.

USER
Indicates that only user-maintained materialized query tables that are refresh deferred will be considered.

host-variable
A variable of type CHAR or VARCHAR. The length of the contents of *host-variable* must not exceed 255 bytes. It cannot be set to null. If *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value.

The characters of *host-variable* must be left justified. The content of the host variable must be a string that would match what can be specified as keywords for the special register in the exact case intended as there is no conversion to uppercase characters.

Notes

The CURRENT REFRESH AGE special register needs to be set to a value other than zero in order for the specified types of objects to be considered for optimizing the processing of dynamic SQL queries.

The CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register affects dynamic statement cache matching.

Examples

Example 1: The following statement sets the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register:

```
SET CURRENT MAINTAINED TABLE TYPES ALL;
```

Example 2: The following example retrieves the current value of the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register into the host variable called CURMAINTYPES.

```
EXEC SQL VALUES (CURRENT MAINTAINED TABLE TYPES) INTO :CURMAINTYPES;
```

The value would be ALL if set by the previous example.

Example 3: The following example resets the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register so that no materialized query tables can be considered to optimize the processing of dynamic SQL queries.

```
SET CURRENT MAINTAINED TABLE TYPES NONE;
```

Related reference:

"CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION" on page 179

SET CURRENT OPTIMIZATION HINT

The SET CURRENT OPTIMIZATION HINT statement assigns a value to the CURRENT OPTIMIZATION HINT special register.

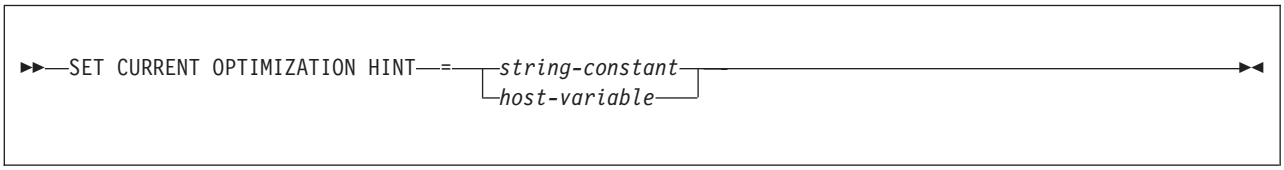
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

The value of special register CURRENT OPTIMIZATION HINT is replaced by the value of the string constant or host variable. The value must be a character string that is not longer than 128 bytes.

Notes

Using the OPTIMIZATION HINT special register: The CURRENT OPTIMIZATION HINT special register specifies whether optimization hints are used in determining the access path of dynamic statements. An empty string or all blanks indicates that DB2 uses normal optimization techniques and ignores optimization hints.

Example

Example 1: Assume that string constant 'NOHYB' identifies a user-defined optimization hint in owner.PLAN_TABLE. Set the CURRENT OPTIMIZATION HINT special register so that DB2 uses this optimization hint to generate the access path for dynamic statements.

```
SET CURRENT OPTIMIZATION HINT = 'NOHYB';
```

If you set the register this way, DB2 validates and considers information in the rows in owner.PLAN_TABLE where the value in the OPTHINT column matches 'NOHYB' for dynamic SQL statements.

Example 2: Clear the CURRENT OPTIMIZATION HINT special register by specifying an empty string.

```
SET CURRENT OPTIMIZATION HINT = '';
```

Related reference:

"CURRENT OPTIMIZATION HINT" on page 181

SET CURRENT PACKAGE PATH

The SET CURRENT PACKAGE PATH statement assigns a value to the CURRENT PACKAGE PATH special register.

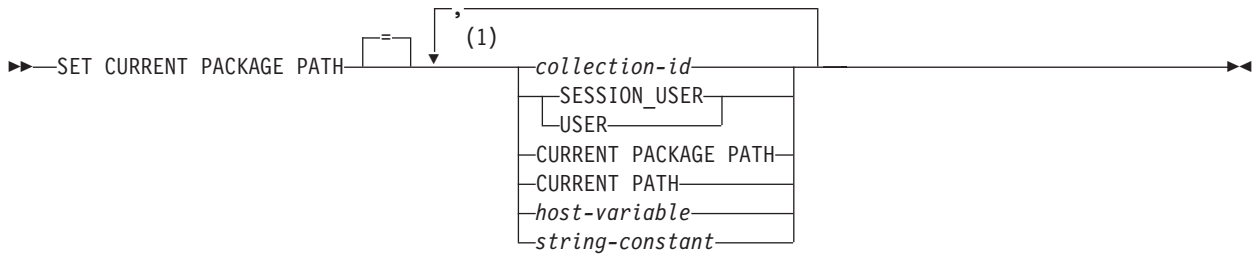
Invocation

This statement can be embedded only in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

None required.

Syntax



Notes:

1 SESSION_USER (or USER), CURRENT PACKAGE PATH, and CURRENT PATH can each be specified only once on the right side of the statement.

Description

The value of CURRENT PACKAGE PATH is replaced by the values specified.

collection-id
Identifies a collection. *collection-id* must not be a delimited identifier that is empty or contains only blanks.

SESSION_USER or USER
Specifies the value of the SESSION_USER (USER) special register.

CURRENT PACKAGE PATH
Specifies the value of the CURRENT PACKAGE PATH special register before the execution of the SET CURRENT PACKAGE PATH statement.

CURRENT PATH
Specifies the value of the CURRENT PATH special register.

host-variable
Specifies a host variable that contains one or more collection IDs, separated by commas. The host variable must:

- Have a data type of CHAR or VARCHAR. The actual length of the contents of the host variable must not exceed the maximum length of the CURRENT PACKAGE PATH special register.
- Not be the null value if an indicator variable is provided.

- Contain an empty or blank string, or one or more collection IDs that are separated by commas.
- Be padded on the right with blanks if the host variable is fixed-length, or if the actual length of the host variable is longer than the content.
- Not contain a delimited identifier that is empty or contains only blanks.

string-constant
Specifies a string constant that contains one or more collection IDs, separated by commas. The string constant must:

- Have a length that does not exceed the maximum length of the CURRENT PACKAGE PATH special register.
- Contain an empty or blank string, or one or more collection IDs separated by commas.
- Not contain a delimited identifier that is empty or contains only blanks.

Notes

Contents of host variable or string constant: The contents of a host variable or string constant are interpreted as a list of collection IDs if the value contains at least one comma. If multiple collection IDs are specified, they must be separated by commas. Each collection ID in the list must conform to the rules for forming an ordinary identifier or be specified as a delimited identifier.

Checking for the existence of collections: No validation that the collections exist is made at the time that the CURRENT PACKAGE PATH special register is set. For example, a collection ID that is misspelled is not detected, which could affect the way subsequent SQL operates. At package execution time, authorization to the specific package is checked, and if this authorization check fails, an error is issued.

Resulting contents of the special register: The special register string is built by taking each collection ID specified and removing trailing blanks, delimiting with double quotation marks, doubling any double quotation marks within the collection ID as necessary, and then separating each collection ID by a comma. If the same collection ID appears more than once in the list, the first occurrence of the collection is used, and a warning is issued. The length of the resulting list cannot exceed the length of the special register. For example, assume that the following statements are issued:

```
SET CURRENT PACKAGE PATH = MYPKGS, "ABC E", SYSIBM
SET :HVPKLIST = CURRENT PACKAGE PATH
```

These statements result in the value of the host variable being set to: "MYPKGS", "ABC E", "SYSIBM".

A collection ID that does not conform to the rules for an ordinary identifier must be specified as a delimited collection ID and must not be specified within a host variable or string constant.

Considerations for keywords: A difference exists between specifying a single keyword, such as SESSION_USER, as a single keyword or as a delimited identifier. To indicate that the current value of a special register that is specified as a single keyword should be used in the package path, specify the name of the special register as a keyword. If you specify the name of the special register as a delimited identifier, it is interpreted as a collection ID of that value. For example, assume that the current value of the SESSION_USER special register is SMITH and that the following statement is issued:

```
SET CURRENT PACKAGE PATH = SYSIBM, SESSION_USER, "USER"
```

The result is that the value of the CURRENT PACKAGE PATH special register is set to: "SYSIBM, "SMITH", "USER".

Specifying a collection ID in an SQL procedure: Because a host variable (SQL variable) in an SQL procedure does not begin with a colon, DB2 uses the following rules to determine whether a value that is specified in a SET PACKAGE PATH = *name* statement is a variable or a collection ID:

- If *name* is the same as a parameter or SQL variable in the SQL procedure, DB2 uses *name* as a parameter or SQL variable and assigns the value in *name* to the package path.
- If *name* is not the same as a parameter or SQL variable in the SQL procedure, DB2 uses *name* as a collection ID and assigns and the value in *name* is the package path.

DRDA classification: The SET CURRENT PACKAGE PATH statement is executed by the database server and, therefore, is classified as a non-local SET statement in DRDA. The SET CURRENT PACKAGE PATH statement requires a new level of DRDA support. If SET CURRENT PACKAGE PATH is issued when connected to the local server, the SET CURRENT PACKAGE PATH special register at the local server is set. Otherwise, when SET CURRENT PACKAGE PATH is issued when connected to a remote server, the SET CURRENT PACKAGE PATH special register at the remote server is set.

Examples

Example 1: Set the CURRENT PACKAGE PATH special register to the list of collections COLL4 and COLL5, where :hvar1 contains the value COLL4,COLL5:

```
SET CURRENT PACKAGE PATH :hvar1;
```

The value of CURRENT PACKAGE PATH is set to the following two collection IDs: "COLL4","COLL5".

Example 2: Set the CURRENT PACKAGE PATH special register to the list of collections: COLL1, COLL#2, COLL3, COLL4, and COLL5, where :hvar1 contains the value COLL4,COLL5:

```
SET CURRENT PACKAGE PATH = "COLL1","COLL#2","COLL3", :hvar1;
```

The value of CURRENT PACKAGE PATH is set to the following five collection IDs: "COLL1,"COLL#2","COLL3","COLL4","COLL5".

Example 3: Clear the CURRENT PACKAGE PATH special register.

```
SET CURRENT PACKAGE PATH = ' ';
```

Example 4: In preparation of calling a stored procedure that is named SUMARIZE, temporarily add two collections, COLL_PROD1" and "COLL_PROD2, to the end of the CURRENT PACKAGE PATH special register (the values of the collections are in host variables :prodcoll1 and prodcoll2, respectively). Because the stored procedure SUMARIZE is not defined with a COLLID value and is defined with INHERIT SPECIAL REGISTERS, the stored procedure will inherit the value of CURRENT PACKAGE PATH. When the stored procedure returns, set the value of the CURRENT PACKAGE PATH special register back to its original value.

```
SET :oldCPP = CURRENT PACKAGE PATH;
SET CURRENT PACKAGE PATH = CURRENT PACKAGE PATH, :prodcoll1, :prodcoll2;
CALL SUMARIZE(:V1,:V2);
SET CURRENT PACKAGE PATH = :oldCPP;
```

Related reference:

"CURRENT PACKAGE PATH" on page 182

SET CURRENT PACKAGESET

The SET CURRENT PACKAGESET statement assigns a value to the CURRENT PACKAGESET special register.

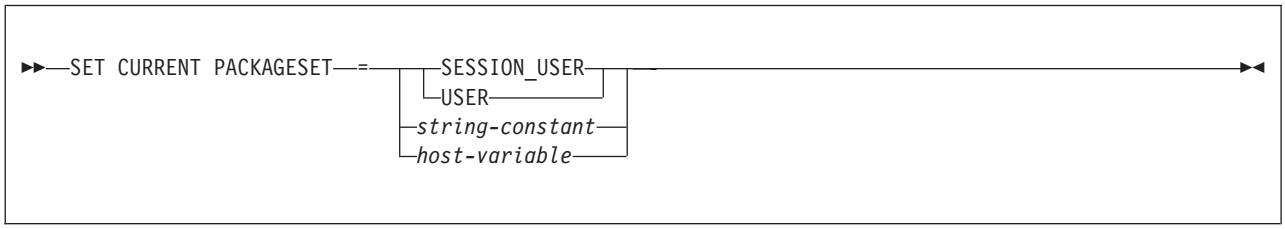
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

None required.

Syntax



Description

The value of CURRENT PACKAGESET is replaced by the value of the SESSION_USER special register, *string-constant*, or *host-variable*. The value specified by *string-constant* or *host-variable* must be a character string that is not longer than 128 bytes.

Notes

Selection of plan elements: A *plan element* is a DBRM that has been bound into the plan or a package that is implicitly or explicitly identified in the package list of the plan. Plan elements contain the control structures used to execute certain SQL statements.

Since a plan can have many elements, one of the first steps involved in the execution of an SQL statement that requires a control structure is the selection of the plan element that contains its control structure. The information used by DB2 to select plan elements includes the value of CURRENT PACKAGESET.

SET CURRENT PACKAGESET is used to specify the collection ID of a package that exists at the current server. SET CURRENT PACKAGESET is optional and should not be used without an understanding of the following rules for selecting a plan element.

If the CURRENT PACKAGESET special register is an empty string, DB2 searches for a DBRM or a package in one of these sequences:

At the local location (if CURRENT SERVER is blank or explicitly names that location), the order is:

1. All DBRMs bound directly to the plan
2. All packages that have already been allocated for the application process

3. All unallocated packages explicitly named in, and all collections completely included in, the package list of the plan. The order of search is the order those packages are named in the package list.

At a remote location, the order is:

1. All packages that have already been allocated for the application process at that location
2. All unallocated packages explicitly named in, and all collections completely included in, the package list of the plan, whose locations match the value of CURRENT SERVER. The order of search is the order those packages are named in the package list.

If the special register CURRENT PACKAGESET is set, DB2 skips the check for programs that are part of the plan and uses the value of CURRENT PACKAGESET as the collection. For example, if CURRENT PACKAGESET contains COL5, then DB2 uses COL5.PROG1.*timestamp* for the search. For additional information, see *DB2 Application Programming and SQL Guide*.

DRDA classification: SET CURRENT PACKAGESET is executed by the requester and is therefore classified as a local SET statement in DRDA.

CURRENT PACKAGESET special register with stored procedures and user-defined functions: The initial value of the CURRENT PACKAGESET special register in a stored procedure or user-defined function is the value of the COLLID parameter with which the stored procedure or user-defined function was defined. If the routine was defined without a value for the COLLID parameter, the value of the special register is inherited from the calling program. A stored procedure or user-defined function can use the SET CURRENT PACKAGESET statement to change the value of the special register. This allows the routine to select the version of the DB2 package that is used to process the SQL statements in a called routine that is not defined with a COLLID value.

When control returns from the stored procedure to the calling program, the special register CURRENT PACKAGESET is restored to the value it contained before the stored procedure was called.

Examples

Example 1: Limit the plan element selection to packages in the PERSONNEL collection at the current server.

```
EXEC SQL SET CURRENT PACKAGESET = 'PERSONNEL';
```

Example 2: Eliminate collections as a factor in plan element selection.

```
EXEC SQL SET CURRENT PACKAGESET = '';
```

Related reference:

"CURRENT PACKAGESET" on page 183

SET CURRENT PRECISION

The SET CURRENT PRECISION statement assigns a value to the CURRENT PRECISION special register.

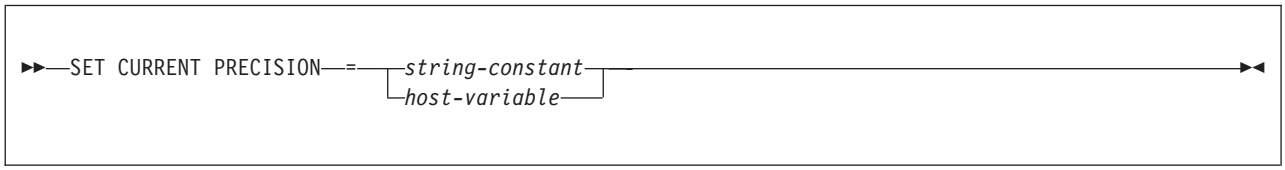
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

This statement replaces the value of the CURRENT PRECISION special register with the value of the string constant or host variable. The value must be a character string 5 bytes in length. The value must be 'DEC15,' 'DEC31,' or 'Dpp.s', where 'pp' is either 15 or 31 and 's' is a number between 1 and 9. If the form 'Dpp.s' is used, 'pp' represents the precision that will be used with the rules that are used for DEC15 or DEC31, and 's' represents the minimum divide scale to use for division operations. The separator used in the form 'Dpp.s' can be either the '.'or the ',' character, regardless of the setting of the default decimal point.

Example

Set the CURRENT PRECISION special register so that subsequent statements that are prepared use DEC15 rules for decimal arithmetic.

```
EXEC SQL SET CURRENT PRECISION = 'DEC15';
```

Related reference:

"CURRENT PRECISION" on page 185

SET CURRENT QUERY ACCELERATION

The SET CURRENT QUERY ACCELERATION statement changes the value of the CURRENT QUERY ACCELERATION special register.

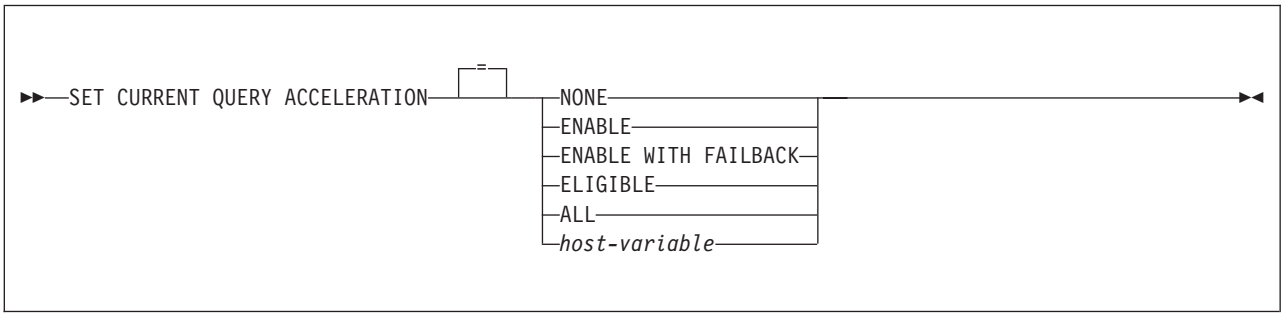
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

NONE
Specifies that no query acceleration is done.

ENABLE
Specifies that queries are accelerated only if DB2 determines that it is advantageous to do so. If an accelerator failure occurs while a query is running or if the accelerator returns an error, DB2 returns a negative SQLCODE to the application.

ENABLE WITH FAILBACK
Specifies that queries are accelerated only if DB2 determines that it is advantageous to do so. If the accelerator returns an error during the PREPARE or first OPEN for the query, DB2 executes the query without the accelerator. If the accelerator returns an error during the PREPARE or first OPEN for the query, DB2 executes the query without the accelerator. If the accelerator returns an error during a FETCH or a subsequent OPEN, DB2 returns the error to the user and does not execute the query.

ELIGIBLE
Specifies that queries are accelerated if they are eligible for acceleration. DB2 does not use cost information to determine whether to accelerate the queries. Queries that are not eligible for acceleration are executed by DB2. If an accelerator failure occurs while a query is running or if the accelerator returns an error, DB2 returns a negative SQLCODE to the application.

ALL
Specifies that queries are accelerated if they are eligible for acceleration. DB2 does not use cost information to determine whether to accelerate the queries. Queries that are not eligible for acceleration are not executed by DB2, and an

SQL error is returned. If an accelerator failure occurs while a query is running or if the accelerator returns an error, DB2 returns a negative SQLCODE to the application.

host-variable
A variable with a data type of CHAR or VARCHAR. The length must not exceed 255 bytes. Valid values are NONE, ENABLE, or ENABLE WITH FAILBACK. If *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value. The value of *host-variable* must be left justified and must be padded on the right with blanks.

Examples

The following statement sets the CURRENT QUERY ACCELERATION special register to NONE to indicate that no acceleration is done.

```
SET CURRENT QUERY ACCELERATION NONE;
```

Related reference:

"CURRENT QUERY ACCELERATION" on page 186

SET CURRENT REFRESH AGE

The SET CURRENT REFRESH AGE statement changes the value of the CURRENT REFRESH AGE special register.

The CURRENT REFRESH AGE value corresponding to ANY (99 999 999 999 999) cannot be used in timestamp arithmetic operations because the result would be outside the valid range of dates.

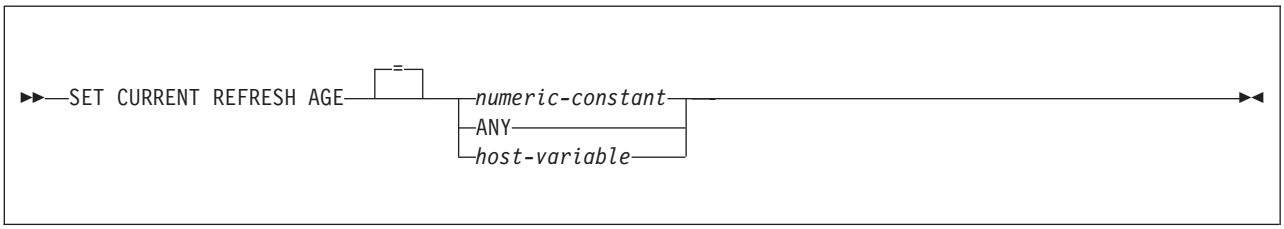
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

numeric-constant
A DECIMAL(20,6) value representing a timestamp duration. The value must be 0 or 99 999 999 999 999, the partial seconds of which is ignored and thus can be any value.

0
Indicates that query optimization using materialized query tables will not be attempted.

99999999999999
Indicates that any materialized query tables identified by the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register may be used to optimize the processing of a query. This value represents 9999 years, 99 months, 99 days, 99 hours, 99 minutes, and 99 seconds.

ANY
Shorthand for 99999999999999.

host-variable
A variable of type DECIMAL(20,6) or other type that is assignable to DECIMAL(20,6). It cannot be set to null. If *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value. The value of *host-variable* must be 0 or 99 999 999 999 999, the partial seconds of which is ignored and thus can be any value.

Notes

Materialized query tables created or altered with DISABLE QUERY OPTIMIZATION specified are not eligible for automatic query rewrite. Thus, they are not affected by the setting of this special register.

Setting the CURRENT REFRESH AGE special register to a value other than zero should be done with caution. Allowing a materialized query table that may not represent the values of the underlying base table to be used to optimize the processing of a query may produce results that do not accurately represent the data in the underlying table. This situation may be acceptable when you know the underlying data has not changed or you are willing to accept the degree of error in the results based on your knowledge of the data.

Examples

Example: Set the CURRENT REFRESH AGE special register to 99 999 999 999 999 to indicate that any materialized query tables identified by the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register can be used to optimize the processing of a query.

```
SET CURRENT REFRESH AGE ANY;
```

Related reference:

"CURRENT REFRESH AGE" on page 187

SET CURRENT ROUTINE VERSION

The SET CURRENT ROUTINE VERSION statement assigns a value to the CURRENT ROUTINE VERSION special register. The special register sets the override value for the version identifier of native SQL procedures when they are invoked.

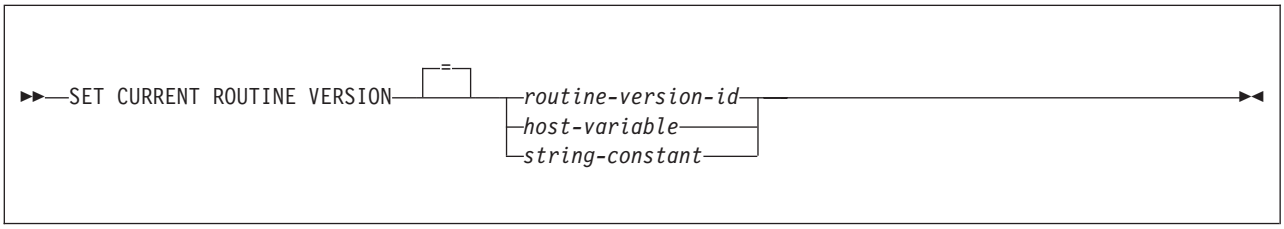
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

routine-version-id
Specifies a routine version identifier.

host-variable
Specifies a host variable that contains a version identifier. The host variable must conform to the following rules:

- Be a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC variable. The actual length of the contents of the host variable must not exceed the length of a version identifier.
- Include a routine version identifier that is left justified and conforms to the rules for forming an ordinary identifier or a delimited identifier, or must be blank or empty.
- Be padded on the right with blanks if the host variable is a fixed length character.
- Not be empty or contain only blanks if the identifier is delimited.
- Not be the null value.

string-constant
Specifies a string constant that contains a version identifier. The string constant must conform to the following rules:

- Have a length that does not exceed the length of a *routine-version-id*.
- Include a routine version identifier that is left justified and conforms to the rules for forming an ordinary identifier or a delimited identifier, or must be blank or an empty string
- Not be empty or contain only blanks if the identifier is delimited

Notes

Resetting the special register: To reset the special register, specify an empty string constant, a string of blanks, or a host variable that is empty or contains only blanks. A routine version override is not in effect when the special register is reset.

Implications of using the special register: Setting the CURRENT ROUTINE VERSION special register to a version identifier will affect all SQL procedures that are subsequently invoked using CALL statements that specify the name of the procedure using a host variable, until the value of CURRENT ROUTINE VERSION is changed. If a version of the procedure that is identified by the version identifier in the special register exists for an SQL procedure that is being invoked, that version of the procedure is used. Otherwise, the currently active version of the procedure (as noted in the catalog) is used.

When you use the CURRENT ROUTINE VERSION special register to test a version of one or more native SQL procedures, you should use a routine version identifier that is a value other than the default value (V1) on the CREATE PROCEDURE statement. This will avoid having the special register affect more procedures that you intend when testing a new version of a procedure. For example, assume that you want to run version VER2 of procedure P1, and procedure P1 invokes another procedure, P2. If a version exists for both procedures P1 and P2 with the routine version identifier VER2, that version will be used for both procedures.

Examples

Example: The following statement sets the CURRENT ROUTINE VERSION special register so that the override value for the version identifier of native SQL procedures will be the value that is specified in the host variable *rvid*:

```
SET CURRENT ROUTINE VERSION = :rvid;
```

Related reference:

"CURRENT ROUTINE VERSION" on page 188

SET CURRENT RULES

The SET CURRENT RULES statement assigns a value to the CURRENT RULES special register.

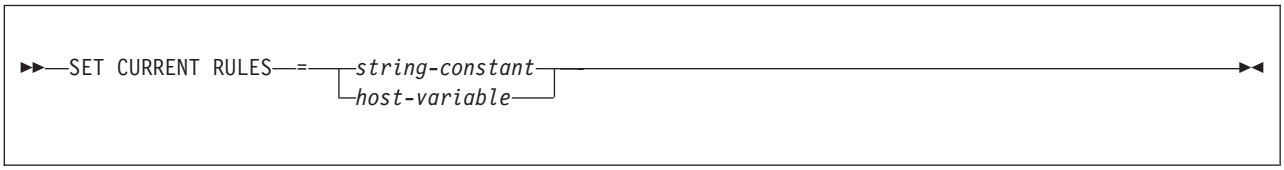
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

This statement replaces the value of the CURRENT RULES special register with the value of the string constant or host variable. The value must be a character string that is 3 bytes in length, and the value must be 'DB2' or 'STD'.

Notes

For the effect of the values 'DB2' and 'STD' on the execution of certain SQL statements, see “CURRENT RULES” on page 189.

Example

Set the SQL rules to be followed to DB2.

```
EXEC SQL SET CURRENT RULES = 'DB2';
```

Related reference:

“CURRENT RULES” on page 189

SET CURRENT SQLID

The SET CURRENT SQLID statement assigns a value to the CURRENT SQLID special register.

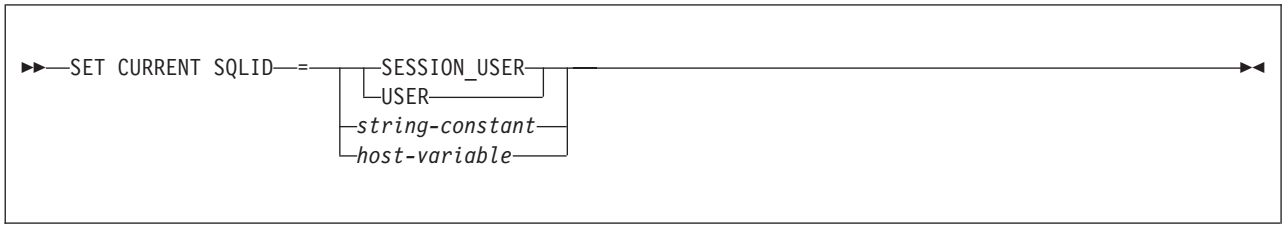
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared. The value to which special register CURRENT SQLID is set is used as the SQL authorization ID for dynamic SQL statements only if DYNAMICRULES run behavior is in effect. The CURRENT SQLID value is ignored for the other DYNAMICRULES behaviors.

Authorization

If any of the authorization IDs of the process has SYSADM authority, CURRENT SQLID can be set to any value when the system parameter, SEPARATE SECURITY, is set to NO. Otherwise, the specified value must be equal to one of the authorization IDs of the application process. This rule always applies, even when SET CURRENT SQLID is a static statement.CURRENT SQLID cannot be set to the name of a role.

Syntax



Description

The value of CURRENT SQLID is replaced by the value of SESSION_USER, *string-constant*, or *host-variable*. The value specified by a *string-constant* or *host-variable* must be a character string that contains 8 characters or less. Unless some authorization ID of the process has SYSADM authority, the value must be equal to one of the authorization IDs of the process.

Notes

Effect on authorization IDs: SET CURRENT SQLID does not change the primary authorization ID of the process.

If the SET CURRENT SQLID statement is executed in a stored procedure or user-defined function package that has a dynamic SQL behavior other than run behavior, the SET CURRENT SQLID statement does not affect the authorization ID that is used for dynamic SQL statements in the package. The dynamic SQL behavior determines the authorization ID. For more information, see the discussion of DYNAMICRULES in *DB2 Command Reference*.

Effect on special register CURRENT PATH: When the value of the PATH special register depends on the value of the CURRENT SQLID special register, any changes to the CURRENT SQLID special register are not reflected in the value of

the PATH special register until a commit operation is performed or a SET PATH statement is issued to change the SQL path to use the new value of the CURRENT SQLID.

DRDA classification: SET CURRENT SQLID is executed by the database server and is therefore classified as a non-local SET statement in DRDA.

Examples

Example 1: Set the CURRENT SQLID to the primary authorization ID.

```
SET CURRENT SQLID = SESSION_USER;
```

SET CURRENT TEMPORAL BUSINESS_TIME

The SET CURRENT TEMPORAL BUSINESS_TIME statement changes the value of the CURRENT TEMPORAL BUSINESS_TIME special register.

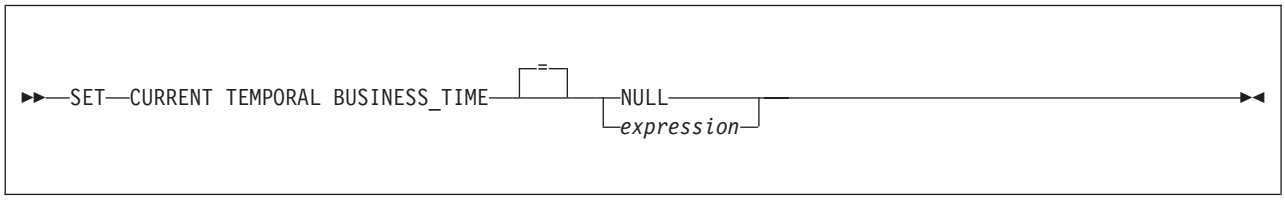
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

NULL
Specifies the null value.

expression
Specifies an expression that returns the null value or the value of one of the following built-in data types:
- Timestamp
- Character string
- Graphic string

If the expression is a character or graphic string, it must meet the following requirements:
- It must not be a CLOB or DBCLOB.
- The value of the expression must be a valid character-string or graphic-string representation of a timestamp.
- The result of the expression must be castable to TIMESTAMP(12).

expression can contain any of the following supported operands:
- Constant
- Special register
- Variable (host variable, SQL parameter, SQL variable, or global variable)
- Scalar function whose arguments are supported operands
- CAST specification where the cast operand is a supported operand
- Expression that uses arithmetic operators and operands

Related information:

"String representations of datetime values" on page 101

"Casting between data types" on page 111

Notes

Transactions

The SET CURRENT TEMPORAL BUSINESS_TIME statement is not a committable operation. The ROLLBACK statement has no effect on CURRENT TEMPORAL BUSINESS_TIME.

Effects on other special registers

The setting of the CURRENT TEMPORAL BUSINESS_TIME special register does not affect other special registers, such as the CURRENT DATE and CURRENT TIMESTAMP special registers.

Examples

Example of setting the special register to a valid value

Both of the following statements set the CURRENT TEMPORAL BUSINESS_TIME special register to '2008-01-06-00.00.00.000000000000'.

```
SET CURRENT TEMPORAL BUSINESS_TIME = TIMESTAMP('2008-01-01') + 5 DAYS ;
SET CURRENT TEMPORAL BUSINESS_TIME = '2008-01-06-00.00.00.000000000000';
```

Example of how setting the special register affects subsequent SQL statements

In the following example, the first statement sets the CURRENT TEMPORAL BUSINESS_TIME special register to last month. Assume that table att1 is an application-period temporal table. The setting of the CURRENT TEMPORAL BUSINESS_TIME special register affects the update of att1.

```
SET CURRENT TEMPORAL BUSINESS_TIME = CURRENT TIMESTAMP - 1 MONTH
UPDATE att1 SET c1 = 5 WHERE pk = 100
```

Assume that the att1 table has columns bt_begin and bt_end to indicate the beginning and end of the BUSINESS_TIME period. In this example, DB2 interprets the UPDATE statement as follows:

```
UPDATE att1 SET c1 = 5 WHERE pk = 100
AND bt_begin <= CURRENT TEMPORAL BUSINESS_TIME
AND bt_end > CURRENT TEMPORAL BUSINESS_TIME
```

Example of setting the special register so that it does not affect subsequent SQL statements

The following statement sets the CURRENT TEMPORAL BUSINESS_TIME special register to the null value. Subsequent SQL statements that reference application-period temporal tables are not affected by the CURRENT TEMPORAL BUSINESS_TIME special register.

```
SET CURRENT TEMPORAL BUSINESS_TIME = NULL
```

Related concepts:

"Data types" on page 80

Related reference:

"CURRENT TEMPORAL BUSINESS_TIME" on page 194

SET CURRENT TEMPORAL SYSTEM_TIME

The SET CURRENT TEMPORAL SYSTEM_TIME statement changes the value of the CURRENT TEMPORAL SYSTEM_TIME special register.

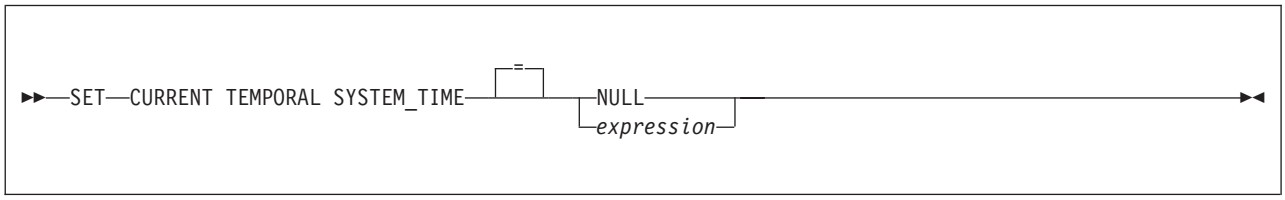
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

NULL
Specifies the null value.

expression
Specifies an expression that returns the null value or the value of one of the following built-in data types:
- Timestamp
- Character string
- Graphic string

If the expression is a character or graphic string, it must meet the following requirements:
- It must not be a CLOB or DBCLOB.
- The value of the expression must be a valid character-string or graphic-string representation of a timestamp.
- The result of the expression must be castable to TIMESTAMP(12).

expression can contain any of the following supported operands:
- Constant
- Special register
- Variable (host variable, SQL parameter, SQL variable, or global variable)
- Scalar function whose arguments are supported operands
- CAST specification where the cast operand is a supported operand
- Expression that uses arithmetic operators and operands

Related information:

"String representations of datetime values" on page 101

"Casting between data types" on page 111

Notes

Transactions

The SET CURRENT TEMPORAL SYSTEM_TIME statement is not a committable operation. The ROLLBACK statement has no effect on CURRENT TEMPORAL SYSTEM_TIME.

Effects on other special registers

The setting of the CURRENT TEMPORAL SYSTEM_TIME special register does not affect other special registers, such as the CURRENT DATE and CURRENT TIMESTAMP special registers.

Examples

Example of setting the special register to a valid value

Both of the following statements set the CURRENT TEMPORAL SYSTEM_TIME special register to '2008-01-06-00.00.00.000000000000'.

```
SET CURRENT TEMPORAL SYSTEM_TIME = TIMESTAMP('2008-01-01') + 5 DAYS;
SET CURRENT TEMPORAL SYSTEM_TIME = '2008-01-06-00.00.00.000000000000';
```

Example of setting the special register so that it does not affect subsequent SQL statements

The following statement sets the CURRENT TEMPORAL SYSTEM_TIME special register to the null value. Subsequent SQL statements that reference system-period temporal tables are not affected by the CURRENT TEMPORAL SYSTEM_TIME special register.

```
SET CURRENT TEMPORAL SYSTEM_TIME = NULL
```

Related concepts:

"Data types" on page 80

Related reference:

"CURRENT TEMPORAL SYSTEM_TIME" on page 196

SET ENCRYPTION PASSWORD

The SET ENCRYPTION PASSWORD statement sets the value of the encryption password and, optionally, the password hint. The encryption and decryption built-in functions use this password and password hint for data encryption unless the functions are invoked with an explicitly specified password and hint. The password is not tied to DB2 authentication and is used only for data encryption.

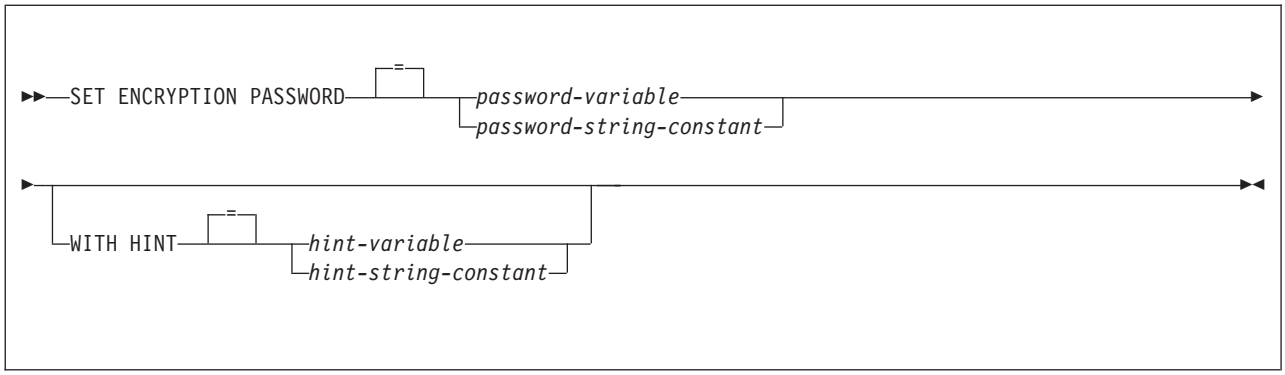
Invocation

The statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

password-variable
Specifies a variable that contains an encryption password. The variable:

- Must be a CHAR or VARCHAR variable. The actual length of the contents of the variable must be between 6 and 127 inclusive or must be an empty string. If an empty string is specified, the default encryption password is set to no value.
- Must not be the null value.
- All characters are case-sensitive and are not converted to uppercase characters.

password-string-constant
A character constant that contains an encryption password. The length of the constant must be between 6 and 127 inclusive or must be an empty string. If an empty string is specified, the default encryption password is set to no value. All characters are case-sensitive and are not converted to uppercase characters.

WITH HINT
Indicates that a value is specified that will help you remember passwords (for example, 'Ocean' as a hint to remember 'Pacific'). If a hint value is specified, the hint is used as the default for encryption functions. The hint can subsequently be retrieved for an encrypted value using the GETHINT function.

If this clause is not specified and a hint is not explicitly specified on the encryption function, no hint will be embedded in encrypted data result.

hint-variable
Specifies a variable that contains an encryption password hint. The variable:

- Must be a CHAR or VARCHAR variable. The actual length of the contents of the variable must not be greater than 32. If an empty string is specified, the default encryption password hint is set to an empty string.
- Must not be the null value.
- All characters are case-sensitive and are not converted to uppercase characters.

hint-string-constant
A character string constant that contains an encryption password hint. The length of the constant must not be greater than 32. If the value is an empty string, the default encryption password hint is set to an empty string.

Notes

Normal DB2 mechanisms are used to transmit the host variable or constant to the database server.

Examples

Example 1: Set the ENCRYPTION PASSWORD to the value in :hv1. Do not specify a hint for the password.

```
SET ENCRYPTION PASSWORD = :hv1
```

Example 2: Set the ENCRYPTION PASSWORD to the value in :hv1. Specify the value in :hv2 as the hint for the password.

```
SET ENCRYPTION PASSWORD = :hv1 WITH HINT :hv2
```

Related reference:

"ENCRYPTION PASSWORD" on page 201

"ENCRYPT_TDES" on page 464

"DECRYPT_BINARY, DECRYPT_BIT, DECRYPT_CHAR, and DECRYPT_DB" on page 451

SET PATH

The SET PATH statement assigns a value to the CURRENT PATH special register.

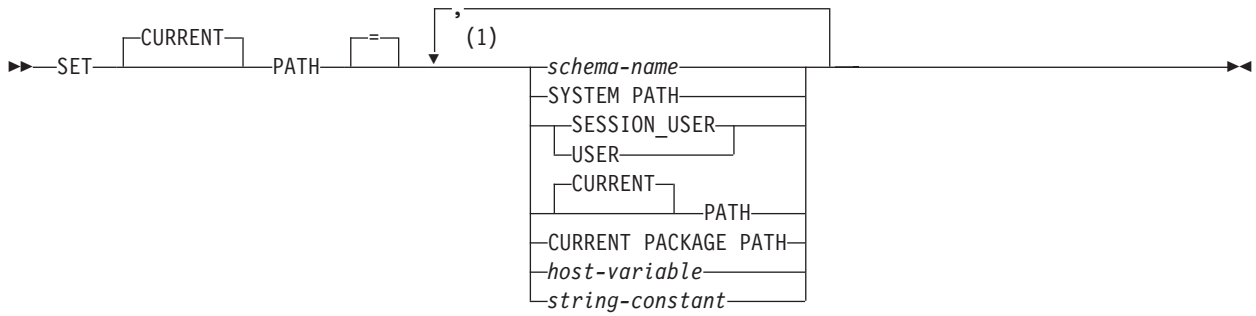
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Notes:

1 SYSTEM PATH, SESSION_USER or USER, and CURRENT PATH can be specified only once each.

Description

The value of PATH is replaced by the values specified.

schema-name
Identifies a schema. DB2 does not verify that the schema exists. For example, a schema name that is misspelled is not detected, which could affect the way subsequent SQL operates.

SYSTEM PATH
Specifies the schema names "SYSIBM", "SYSFUN", "SYSPROC", "SYSIBMADM".

SESSION_USER or USER
Specifies the value of the SESSION_USER (USER) special register.

PATH
Specifies the value of the CURRENT PATH special register before the execution of this statement.

CURRENT PACKAGE PATH
Specifies the value of the CURRENT PACKAGE PATH special register.

host-variable
A variable with a data type of CHAR or VARCHAR. The value of *host-variable* must not be null and must represent a valid schema name.

The schema name must:

- Be left justified within the host variable
- Be padded on the right with blanks if its length is less than that of the host variable

string-constant
A character string constant that represents a valid schema name.

If the schema name specified in *string-constant* will also be specified in other SQL statements and the schema name does not conform to the rules for ordinary identifiers, the schema name must be specified as a delimited identifier in the other SQL statements.

Notes

Restrictions on SET PATH:
These restrictions apply to the SET PATH statement:

- If the same schema name appears more than one time in the path, the first occurrence of the name is used and a warning is issued.
- The length of the CURRENT PATH special register limits the number of schema names that can be specified. The special register string is built by taking each schema name that is specified and removing trailing blanks, delimiting with double quotes, changing each double quote character to two double quote characters within the schema name as necessary, and then separating each schema name with a comma. The length of the resulting string cannot exceed 2048 bytes.
- The schema name SYSPUBLIC cannot be specified in the SQL path, even if you specify the value as a delimited identifier.

Specifying "SYSIBM", "SYSFUN", "SYSPROC", "SYSIBMADM":
Schemas "SYSIBM", "SYSFUN", "SYSPROC", "SYSIBMADM" do not need to be specified in the special register. If these schemas are not explicitly specified in the CURRENT PATH special register, each schema is implicitly assumed at the front of the SQL path; if any of these schemas are not specified, they are assumed in the order of "SYSIBM", "SYSFUN", "SYSPROC", "SYSIBMADM" (see “SQL path” on page 64 for an example). Only the schemas that are explicitly specified in the CURRENT PATH register are included in the 2048 byte limit.

To avoid having "SYSIBM", "SYSFUN", "SYSPROC", "SYSIBMADM" implicitly added to the front of the SQL path, explicitly specify them in the path when setting the value of the register. If you specify them at the end of the path, DB2 will check all the other schemas in the path first.

Specifying keywords versus delimited identifiers:
There is a difference between specifying a keyword and specifying a delimited identifier. For example, specifying SESSION_USER with and without escape characters. To indicate that the value of the SESSION_USER special register should be used in the SQL path, specify the keyword SESSION_USER. If you specify SESSION_USER is as a delimited identifier instead (for example, "SESSION_USER"), it is interpreted as a schema name of 'SESSION_USER'. For example, assume that the current value of the SESSION_USER special register is SMITH and that the following statement is issued:

```
SET PATH = SYSIBM, SYSPROC, SESSION_USER, "SESSION_USER"
```

The result is that the value of the SQL path is set to:
"SYSIBM","SYSPROC","SMITH","SESSION_USER".

Specifying a schema name in an SQL procedure:

Because a host variable (SQL variable) in an SQL procedure does not begin with a colon, DB2 uses the following rules to determine whether a value that is specified in a SET PATH=*name* statement is a variable or a *schema-name*:

- If *name* is the same as a parameter or SQL variable in the SQL procedure, DB2 uses *name* as a parameter or SQL variable and assigns the value in *name* to PATH.
- If *name* is not the same as a parameter or SQL variable in the SQL procedure, DB2 uses *name* as a *schema-name* and assigns the value *name* to PATH.

The use of the path to resolve object names:

For information on when the SQL path is used to resolve unqualified data type, function, and procedure names and when the CURRENT PATH special register provides the SQL path, see "SQL path" on page 64.

DRDA classification:

The SET PATH statement is executed by the database server and, therefore, is classified as a non-local SET statement in DRDA.

Alternative syntax and synonyms:

For compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports CURRENT FUNCTION PATH or CURRENT_PATH as a synonym for CURRENT PATH. CURRENT_PATH is consistent with the SQL standard name of the special register.

Examples

Example 1: Set the CURRENT PATH special register to the list of schemas: "SCHEMA1", "SCHEMA#2", "SYSIBM".

```
SET PATH = SCHEMA1,"SCHEMA#2", SYSIBM;
```

When the SQL path specified in the special register is used for name resolution the system schemas which were not explicitly specified in the special register are implicitly assumed at the front of the SQL path, making the effective value of the path:

```
SYSFUN, SYSPROC, SYSIBMADM, SCHEMA1, SCHEMA#2, SYSIBM
```

Example 2: Add schema SMITH and SYSPROC to the value of the CURRENT PATH special register that was set in Example 1.

```
SET PATH = CURRENT PATH, SMITH, SYSPROC;
```

The effective value of the SQL path specified by the special register becomes:

```
SYSFUN, SYSIBMADM, SCHEMA1, SCHEMA#2, SYSIBM, SMITH, SYSPROC
```

Related reference:

"CURRENT PATH" on page 184

SET SCHEMA

The SET SCHEMA statement assigns a value to the CURRENT SCHEMA special register. If the package is bound with the DYNAMICRULES BIND option, this statement does not affect the qualifier that is used for unqualified database object references.

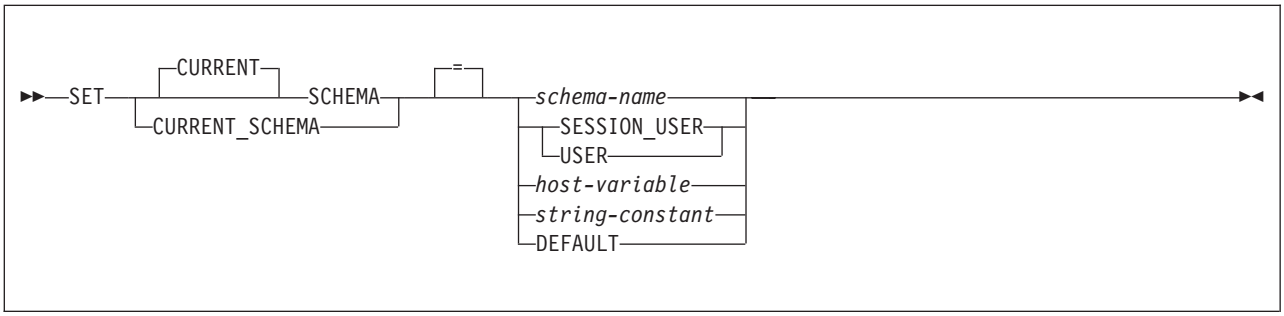
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

schema-name
Identifies a schema. No validation that the schema exists is made at the time the CURRENT SCHEMA is set. For example, if a schema name is misspelled, it could affect the way subsequent SQL operates.

SESSION_USER or USER
Specifies the value of the SESSION_USER (USER) special register.

host-variable
Specifies a host variable that contains a schema name. The content is not folded to uppercase.

The host variable must:

- Be a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC variable. The actual length of the contents of the *host-variable* must not exceed the length of a schema name.
- Not be set to null. If *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value.
- Include a schema name that is left justified and conforms to the rules for forming an ordinary identifier or delimited identifier. If the identifier is delimited, it must not be empty or contain only blanks.
- Be padded on the right with blanks if the host variable is fixed length.
- Not contain SESSION_USER, USER, or DEFAULT.

string-constant
Specifies a string constant that contains a schema name. The content is not folded to uppercase.

The string constant must:

- Have a length that does not exceed the maximum length of a schema name.
- Include a schema name that is left justified and conforms to the rules for forming an ordinary identifier or delimited identifier. If the identifier is delimited, it must not be empty or contain only blanks.
- Not contain SESSION_USER, USER, or DEFAULT.

DEFAULT
Specifies that CURRENT SCHEMA is to be set to its initial value, as if it had never been explicitly set during the application process. For information about the initial value of CURRENT SCHEMA, see "CURRENT SCHEMA" on page 191.

Notes

Considerations for keywords:
There is a difference between specifying a single keyword (such as SESSION_USER or DEFAULT) as a single keyword or as a delimited identifier. To indicate that the current value of the SESSION_USER special register should be used for setting the current schema, specify SESSION_USER as a keyword. To indicate that the special register should be set to its default value, specify DEFAULT as a keyword. If SESSION_USER or DEFAULT is specified as a delimited identifier instead (for example, "SESSION_USER"), it is interpreted as a schema name of that value ("SESSION_USER").

Transaction considerations:
The SET SCHEMA statement is not a committable operation. ROLLBACK has no effect on CURRENT SCHEMA.

Usage of the assigned value:
The value of the CURRENT SCHEMA special register, as set by this statement, is used as the schema name in all dynamic SQL statements. The QUALIFIER bind option specifies the schema name for use as the qualifier for unqualified database object names in static SQL statements.

Impact on other special registers:
Setting the CURRENT SCHEMA special register does not affect any other special register. Therefore, the CURRENT SCHEMA is not be included in the SQL path that is used to resolve the schema name for unqualified references to function, procedures and user-defined types in dynamic SQL statements. To include the current schema value in the SQL path, whenever the SET SCHEMA statement is issued, also issue the SET PATH statement including the schema name from the SET SCHEMA statement.

Examples

Example 1: The following statement sets the CURRENT SCHEMA special register.

```
EXEC SQL SET SCHEMA RICK;
```

Example 2: The following example retrieves the current value of the CURRENT SCHEMA special register into the host variable called CURSCHEMA.

```
EXEC SQL SELECT CURRENT SCHEMA INTO :CURSCHEMA
   FROM SYSIBM.SYSDUMMY1;
```

The value of the host variable is RICK.

Example 3: Assume that the following statements are issued:

```
SET CURRENT SQLID = 'USRT001';
SET CURRENT SCHEMA = 'USRT002';
```

At this point, the two special registers contain different values. Any subsequent CREATE statements will use USRT002 as the implicit qualifier, but the owner of the newly created objects is USRT001.

Example 4: Assume that the value of CURRENT SCHEMA is 'Jane' and that the default value of the PATH special register was established using that value (that is, the value of PATH is "SYSIBM", "SYSFUN", "SYSPROC", "SYSIBMADM", "JANE"). Change the value of the CURRENT SCHEMA special register to 'John'.

```
SET CURRENT SCHEMA = 'JOHN';
```

To change the SQL path to use the updated CURRENT SCHEMA value of "JOHN", issue a SET PATH statement to change the value of the PATH special register to specify "JOHN" as the first schema to check:

```
SET PATH = 'JOHN', CURRENT PATH;
```

Alternatively, a commit would cause PATH to be re-initialized. Otherwise, the path remains "SYSIBM", "SYSFUN", "SYSPROC", "SYSIBMADM", "JANE"), which might cause unqualified object names to resolve to "JANE" when you want them to resolve to "JOHN".

SET SESSION TIME ZONE

The SET SESSION TIME ZONE statement assigns a value to the SESSION TIME ZONE special register.

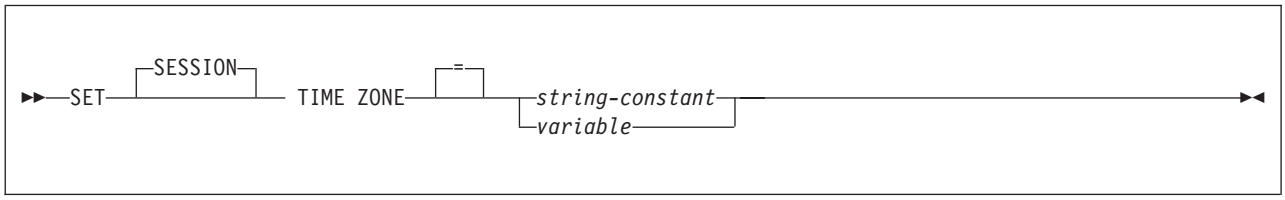
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

string-constant
Identifies a time zone with a value of the form '±*th*:*tm*', where *th* represents the time zone hour between -12 and +14, and *tm* represents the time zone minutes between 0 and 59, with values ranging from -12:59 to +14:00.

variable
Specifies a variable that contains a time zone. The variable must be a CHAR or VARCHAR variable that is not followed by an indicator variable. The variable must not be the null value. The value must be left justified and be of the form '±*th*:*tm*', where *th* represents the time zone hour between -12 and +14, and *tm* represents the time zone minutes between 0 and 59, with values ranging from -12:59 to +14:00.

Notes

Impact on other special registers:
Setting the SESSION TIME ZONE special register does not affect the CURRENT TIMEZONE special register.

Syntax alternatives:
SESSIONTIMEZONE, SESSION TIMEZONE, and TIMEZONE can be specified as an alternative to SESSION TIME ZONE or TIME ZONE.

Example

Set the SESSION TIME ZONE as -8:00:

```
SET SESSION TIME ZONE = '-8:00';
```

Related reference:
"SESSION TIME ZONE" on page 203

SIGNAL

The SIGNAL statement is used to signal an error. It causes an error to be returned with the specified SQLSTATE and error description.

For a description of the statement, see “SIGNAL statement” on page 2006.

TRUNCATE

The DB2 TRUNCATE statement deletes all rows for either base tables or declared global temporary tables. The base table can be in a simple table space, a segmented table space, a partitioned table space, or a universal table space. If the table contains LOB or XML columns, the corresponding table spaces and indexes are also truncated.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privilege set that is defined below must include at least one of the following privileges:

- The DELETE privilege for the table
- Ownership of the table
- DBADM authority for the database
- DATAACCESS authority
- SYSADM authority

If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

If the IGNORE DELETE TRIGGERS option is specified, the privilege set must include at least one of the following privileges:

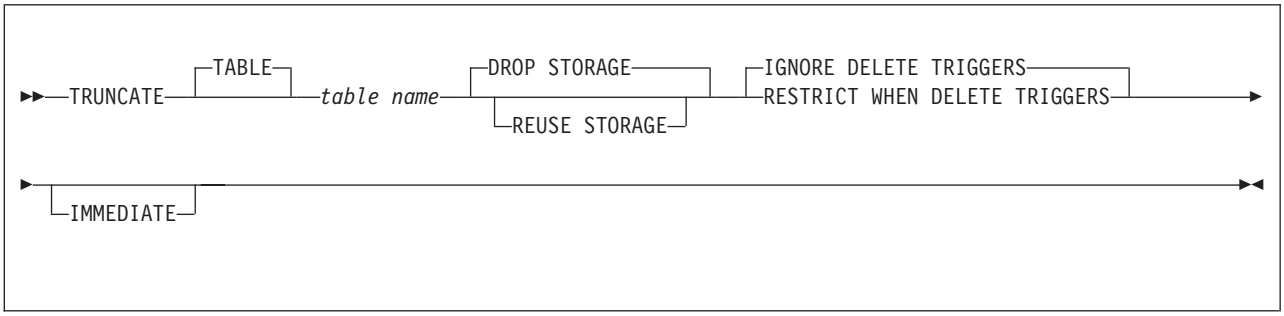
- The ALTER privilege for the table
- Ownership of the table
- DBADM authority for the database
- System DBADM authority
- SYSADM authority

If row access control is activated for the table, the privilege set must include at least one of the following privileges or authorities:

- Ownership of the table
- DBADM authority
- SYSADM authority
- SYSCTRL authority
- System DBADM authority

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set is determined by the DYNAMICRULES behavior in effect (run, bind, define, or invoke) and is summarized in Table 94 on page 841. (For more details on these behaviors, including a list of the DYNAMICRULES bind option values that determine them, see "Authorization IDs and dynamic SQL" on page 75.)

Syntax



Description

table-name
Identifies the table that is to be truncated. The name must identify a table that exists at the current server. The name must not identify the following objects:

- a view
- an auxiliary table
- an XML table
- a catalog table
- a system-period temporal table

If *table-name* is a base table of a table space, all tables that are defined under the table will also be truncated (for example: auxiliary LOB table spaces and XML table spaces), and all of its associated indexes will also be truncated.

DROP STORAGE or REUSE STORAGE
Specifies whether to drop or reuse the existing storage that is allocated for the table.

DROP STORAGE
Specifies that all storage that is allocated for the table is released and made available for use for the same table or any other table that resides in the table space. The scope of **DROP STORAGE** is always at the table space level and the deallocated space is always available for reuse by all tables in the table space.

DROP STORAGE is the default.

REUSE STORAGE
Specifies that all storage that is allocated for the table will be emptied, but will continue to be allocated for the table. **REUSE STORAGE** is ignored for a table in a simple table space and the statement is processed as if **DROP STORAGE** is specified.

RESTRICT WHEN DELETE TRIGGERS or IGNORE DELETE TRIGGERS
Specifies what to do when delete triggers are defined on the table.

RESTRICT WHEN DELETE TRIGGERS
Specifies that an error is returned if delete triggers are defined on the table.

IGNORE DELETE TRIGGERS
Specifies that any delete triggers that are defined for the table are not activated by the truncate operation.

IGNORE DELETE TRIGGERS is the default.

IMMEDIATE

Specifies that the truncate operation is processed immediately and cannot be undone. If the **IMMEDIATE** option is specified, the table must not contain any uncommitted updates. In the case of a table in a multi-table table space, if there are any uncommitted updates to any table in the table space, the truncate operation will fail. Also, if there are any uncommitted CREATE, ALTER or DROP statements for any table in the table space, the truncate operation will fail.

The truncated table is immediately available for use in the same unit of work. Although a ROLLBACK statement is allowed to execute after a TRUNCATE statement, the truncate operation is not undone, and the table remains in a truncated state. For example, if another data change operation is done on the table after the TRUNCATE IMMEDIATE statement and then the ROLLBACK statement is executed, the truncate operation will not be undone, but all other data change operations are undone.

If **IMMEDIATE** is not specified, a ROLLBACK statement can undo the truncate operation.

The **IMMEDIATE** option can be specified for a table in a segmented table space or a universal table space which allows deallocated spaces to be reclaimed immediately for subsequent insert operations in the same unit of work without committing the truncate operation.

Notes

Rules and restrictions:

The truncate operation cannot be executed if the table is a parent table in an enforced referential constraint. The DB2 subsystem issues an error when it detects the existence of rule violations. Therefore, if the referential integrity constraint exists, the TRUNCATE statement will be restricted regardless of whether the child table contains rows.

The TRUNCATE statement cannot be used if the table is a system-maintained temporal table.

If the TRUNCATE statement is used on a tables where any of the following conditions is true, the truncate operation will perform in a similar way to a mass delete operation:

- Tables with Change Data Capture (CDC) attribute

  The DB2 subsystem allows a table with the CDC-enabled attribute to be truncated without imposing any new restrictions.
- Tables with multi-level security

  If the table contains a column that is defined as a security label, the truncate operation needs to examine each row to determine if the security label of the authorization ID or role has the authority to delete that row. However, if the primary authorization ID or role has write-down privilege, verification of each row in the table is not necessary.
- Tables with a VALIDPROC attribute

  If a VALIDPROC is defined for the table, the truncate operation needs to verify the validity of each row in the table.

TRUNCATE and table spaces that are not logged:

The TRUNCATE TABLE statement can be used to remove a table space from the logical page list and to reset recover-pending status. When the table space is segmented or universal, the table is the only table in the

table space, and the table does not have a VALIDPROC, referential constraints, delete triggers, or a SECURITY LABEL column, use the TRUNCATE TABLE statement to empty the table and the table space will be removed from the LPL and recover-pending status will be reset.

Truncating rows from a table with activated row permissions or column access control:

Row permissions and column access control is not enforced for the TRUNCATE statement.

Examples

Example 1: Empty an unused inventory table regardless of any existing triggers and return its allocated space.

```
TRUNCATE TABLE INVENTORY
DROP STORAGE
    IGNORE DELETE TRIGGERS;
```

Example 2: Empty an unused inventory table regardless of any existing delete triggers but preserve its allocated space for later reuse.

```
TRUNCATE TABLE INVENTORY
   REUSE STORAGE
   IGNORE DELETE TRIGGERS;
```

Example 3: Empty an unused inventory table permanently (a ROLLBACK statement cannot undo the truncate operation when the **IMMEDIATE** option is specified) regardless of any existing delete triggers and preserve its allocated space for immediate use.

```
TRUNCATE TABLE INVENTORY
    REUSE STORAGE
    IGNORE DELETE TRIGGERS
    IMMEDIATE;
```

UPDATE

The UPDATE statement updates the values of specified columns in rows of a table or view or activates an instead of update trigger. Updating a row of a view updates a row of the table on which the view is based if no instead of update trigger is defined for the update operation on the view. If such a trigger is defined, the trigger is activated instead of the UPDATE statement. The table or view can exist at the current server or at any DB2 subsystem with which the current server can establish a connection.

There are two forms of this statement:

- The *searched* UPDATE form is used to update one or more rows optionally determined by a search condition.
- The *positioned* UPDATE form specifies that one or more rows corresponding to the current cursor position are to be updated.

Invocation

This statement can be embedded in an application program or issued interactively. A positioned UPDATE can be embedded in an application program. Both forms are executable statements that can be dynamically prepared.

Authorization

Authority requirements depend on whether the object identified in the statement is a user-defined table, a catalog table for which updates are allowed, or a view, and whether SQL standard rules are in effect:

When a user-defined table is identified: The privilege set must include at least one of the following:

- The UPDATE privilege on the table
- The UPDATE privilege on each column to be updated
- Ownership of the table
- DBADM authority on the database that contains the table
- SYSADM authority

If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

When a catalog table is identified: The privilege set must include at least one of the following:

- The UPDATE privilege on each column to be updated
- DBADM authority on the catalog database
- SYSCTRL authority
- SYSADM authority

When a view is identified: The privilege set must include at least one of the following:

- The UPDATE privilege on the view
- The UPDATE privilege on each column to be updated
- SYSADM authority

If the *expression* in the *assignment-clause* contains a reference to a column of the table or view, or if the *search-condition* in a searched UPDATE contains a reference to a column of the table or view, the privilege set must include at least one of the following:

- The SELECT privilege on the table or view
- Ownership of the table or view
- DBADM authority on the database that contains the table, if the target is a table and that table that is not a catalog table
- DATAACCESS
- SYSADM authority

When FOR PORTION OF BUSINESS_TIME is specified: The privilege set must include at least one of the following:

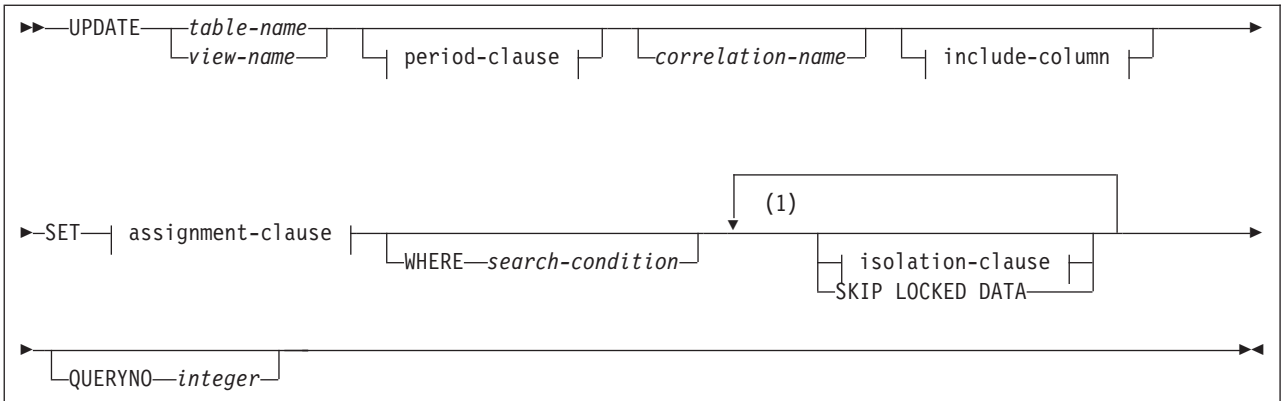
- The UPDATE privilege on the columns of the BUSINESS_TIME period
- The UPDATE privilege on the table
- Ownership of the table or view
- DBADM authority on the database that contains the table, if the target is a table and that table that is not a catalog table
- DATAACCESS
- SYSADM authority

If the *search-condition* in a searched UPDATE includes a subquery, or if the *assignment-clause* includes a *scalar-fullselect* or a *row-fullselect*, see "Authorization" on page 762 for an explanation of the authorization required.

The owner of a view, unlike the owner of a table, might not have UPDATE authority on the view (or might have UPDATE authority without being able to grant it to others). The nature of the view itself can preclude its use for UPDATE. For more information, see the discussion of authority in "CREATE VIEW" on page 1527.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set is determined by the DYNAMICRULES behavior in effect (run, bind, define, or invoke) and is summarized in Table 94 on page 841. (For more information on these behaviors, including a list of the DYNAMICRULES bind option values that determine them, see "Authorization IDs and dynamic SQL" on page 75).

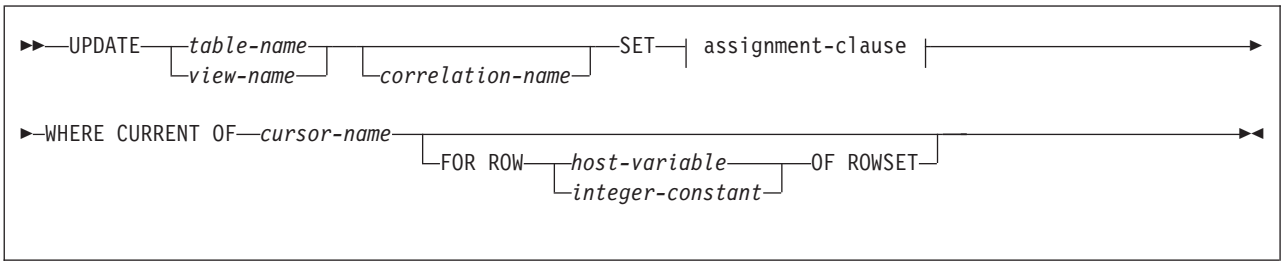
searched update:



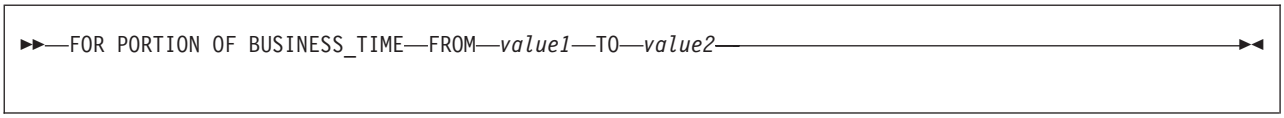
Notes:

1 The same clause must not be specified more than one time.

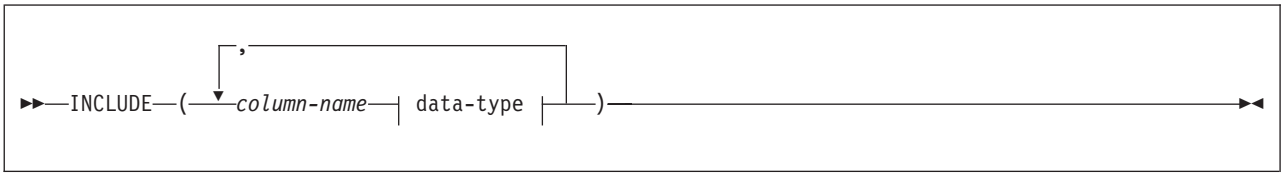
positioned update:



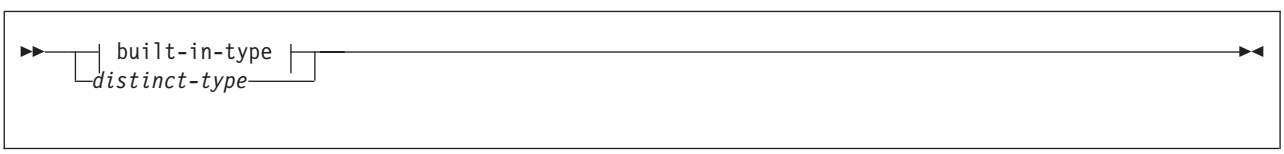
period-clause:



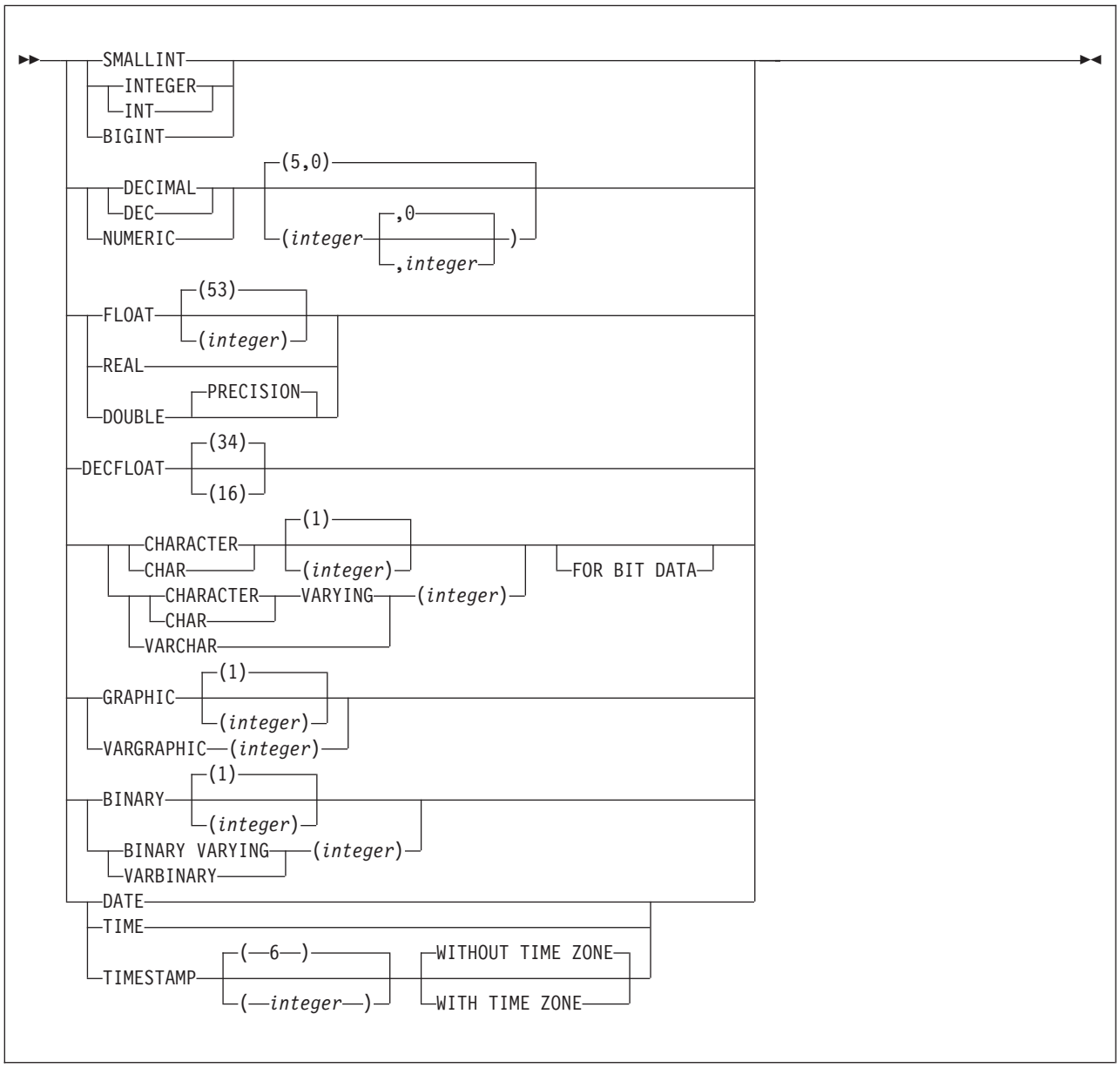
include-column:



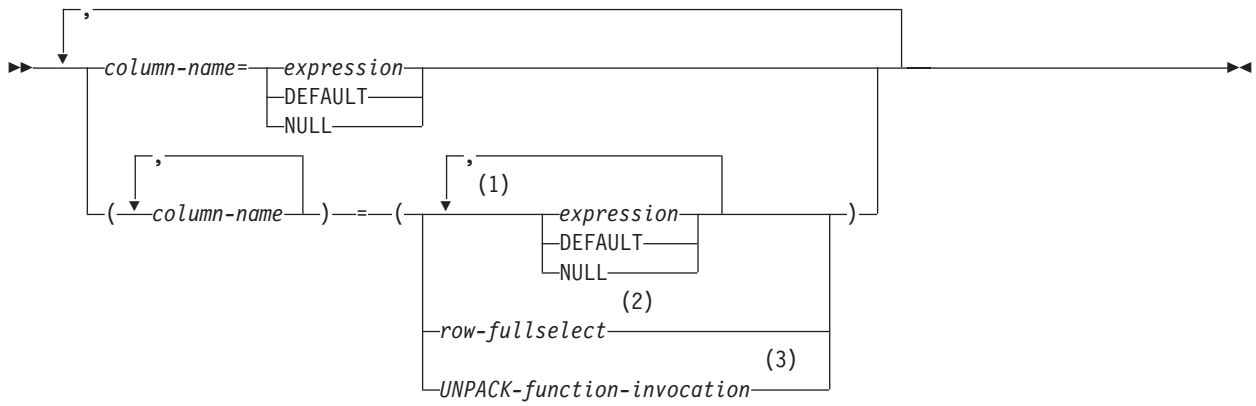
data-type:



built-in-type:



assignment clause:



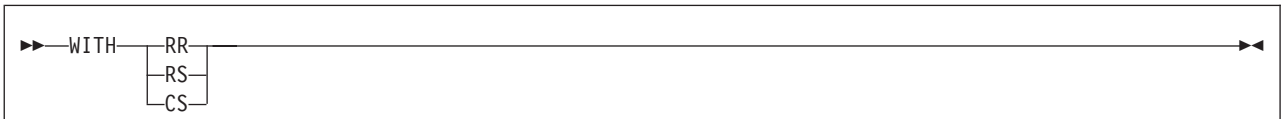
Notes:

1 The number of *expressions*, DEFAULT, and NULL keywords must match the number of *column-names*. *Expressions* must not refer to UNPACK-function-invocation..

2 The number of columns in the select list must match the number of *column-names*.

3 The number of items returned from *UNPACK-function-invocation* must match the number of column names.

isolation-clause:



Description

table-name **or** *view-name*

Identifies the object of the UPDATE statement. The name must identify a table or view that exists at the DB2 subsystem that is identified by the implicitly or explicitly specified location name. The name must not identify one of the following tables:

- An auxiliary table
- A created temporary table or a view of a created temporary table
- A catalog table with no updatable columns or a view of a catalog table with no updatable columns
- A read-only view that has no INSTEAD OF trigger defined for its update operations. (For a description of a read-only view, see "CREATE VIEW" on page 1527.)
- A system-maintained materialized query table
- A table that is implicitly created for an XML column
- An archive-enabled table if any of the following conditions are true:
  - The SYSIBMADM.MOVE_TO_ARCHIVE global variable is set to Y.
  - The SYSIBMADM.GET_ARCHIVE global variable is set to Y, the ARCHIVESENSITIVE bind option is set to YES, and the operation is a positioned update.

In an IMS or CICS application, the DB2 subsystem that contains the identified table or view must be a remote server that supports two-phase commit.

A catalog table or a view of a catalog table can be identified if every column identified in the SET clause is an updatable column. If a column of a catalog table is updatable, its description in "DB2 catalog tables" on page 2102 indicates that the column can be updated. If the object table is SYSIBM.SYSSTRINGS, any column other than IBMREQD can be updated, but the rows that are selected for update must be rows that are provided by the user (the value of the IBMREQD column is N) and only certain values can be specified as explained in *DB2 Administration Guide*.

period-clause
Specifies that a period clause applies to the target of the update operation. The same period name must not be specified more than one time. If the target of the update operation is a view:

- The FROM clause of the outer fullselect of the view definition must include a reference, directly or indirectly, to an application-period temporal table.
- The result table of the outer fullselect of the view definition must include, explicitly or implicitly, the start and end columns of the BUSINESS_TIME period.
- An INSTEAD OF trigger must not be defined for the view.

FOR PORTION OF BUSINESS_TIME
Specifies that the update only applies to row values for the portion of the BUSINESS_TIME period in the row that is specified by the period clause. BUSINESS_TIME must be a period that is defined on the table.

FROM *value1* **TO** *value2*
Specifies that the update applies to rows for the period that is specified from *value1* to *value2*. No rows are updated if *value1* is greater than or equal to *value2* or if *value1* or *value2* is the null value.

For the period condition that is specified with **FROM** *value1* **TO** *value2*, the period that is specified with *period-name* in a row of the target update covers one of the following ranges:

- If the value of the begin column is less than *value1* and the value of the end column is greater than *value1*, the range overlaps the beginning of the specified period.
- If the value of the end column is greater than or equal to *value2* and the value of the begin column is less than *value2*, the range overlaps the end of the specified period.
- If the value for the begin column is greater than or equal to *value1* and the value for the end column is less than or equal to *value2*, the range is fully contained within the specified period.
- If both columns of *period-name* are less than or equal to *value1* or greater than or equal to *value2*, the range is not contained in the period
- If the period in the row overlaps the beginning of the specified period or the end of the specified period, but not both, the range is partially contained in the specified period.
- If the period in the row overlaps both the beginning and the end of the specified period, the range fully overlaps the specified period.

If the period, *period-name* in a row is not contained in the specified period, the row is not updated. Otherwise, the update is applied based on the specification of PORTION OF and how the values in the columns of *period-name* overlap the specified period as follows:

- If the period, *period-name* in a row is fully contained within the specified period, the row is updated and the values of the begin column and end column of *period-name* are unchanged.
- If the period, *period-name* in a row is partially contained in the specified period and overlaps the beginning of the specified period:
  - The row is updated. In the updated row, the value of the begin column is set to *value1* and the value of the end column is the original value of the end column.
  - An additional row is inserted using the original values from the row, except that the end column is set to *value1*.
- If the period, *period-name* in a row is partially contained in the specified period and overlaps the end of the specified period:
  - The row is updated. In the updated row, the value of the begin column is the original value of the begin column and the end column is set to *value2*.
  - An additional row is inserted using the original values from the row, except that the begin column is set to *value2*.
- If the period, *period-name* in a row fully overlaps the specified period:
  - The row is updated. In the updated row, the value of the begin column is set to *value1* and the value of the end column is set to *value2*.
  - An additional row is inserted using the original values from the row, except that the end column is set to *value1* and the begin column is set to *value2*.

Any existing update triggers are activated for the updated rows and any existing insert triggers are activated for rows that are implicitly inserted.

value1, *value2*
Specifies expressions that return a value of a built-in data type. The result of each expression must be comparable to the data type of the columns of the specified period. See the comparison rules described in "Assignment and comparison" on page 121. Each expression can contain any of the following supported operands:

- A constant
- A special register
- A variable (host variable, SQL variable, SQL parameter, or transition variable)
- An array element specification
- A built-in scalar function whose arguments are supported operands
- A CAST specification where the cast operand is a supported operand
- An expression that uses arithmetic operators and operands

Each expression must not have a timestamp precision that is greater than the precision of the columns for the period.

If the begin and end columns of the period are defined as TIMESTAMP WITHOUT TIME ZONE, each expression must not return a value of a timestamp with a time zone.

A period clause for a view must not contain a global variable or an untyped parameter marker.

correlation-name
Can be used within *search-condition* or *assignment-clause* to designate the table or view. (For an explanation of *correlation-name*, see "Correlation names" on page 209.)

include-column
Specifies a set of columns that are included, along with the columns of *table-name* or *view-name*, in the result table of the UPDATE statement when it is nested in the FROM clause of the outer fullselect that is used in a subselect, SELECT statement, or in a SELECT INTO statement. The included columns are appended to the end of the list of columns that is identified by *table-name* or *view-name*. If no value is assigned to a column that is specified by an *include-column*, a NULL value is returned for that column.

INCLUDE
Introduces a list of columns that are to be included in the result table of the UPDATE statement. The included columns are only available if the UPDATE statement is nested in the FROM clause of a SELECT statement or a SELECT INTO statement.

column-name
Specifies the name for a column of the result table of the UPDATE statement that is not the same name as another included column nor a column in the table or view that is specified in *table-name* or *view-name*.

data-type
Specifies the data type of the included column. The included columns are nullable.

built-in-type
Specifies a built-in data type. See "CREATE TABLE" on page 1388 for a description of each built-in type.

The CCSID 1208 and CCSID 1200 clauses must not be specified for an include column.

distinct-type
Specifies a distinct type. Any length, precision, or scale attributes for the column are those of the source type of the distinct type as specified by using the CREATE TYPE statement.

SET
Introduces the assignment of values to column names.

assignment-clause
If *row-fullselect* is specified, the number of columns in the result of *row-fullselect* must match the number of *column-names* that are specified. If *row-fullselect* is not specified, the number of expressions, and NULL and DEFAULT keywords must match the number of *column-names* that are specified.

column-name
Identifies a column that is to be updated. *column-name* must identify a column of the specified table or view, and that column must be updatableif extended indicator variables are not enabled. The column must not identify a generated column or a view column where the column is derived from a scalar function, constant, or expression. *column-name* can also identify an INCLUDE column that must not be qualified. The same column must not be specified more than one time.

A column that is defined as part of a BUSINESS_TIME period must not be specified if the UPDATE statement contains a *period-clause*.

Assignments to included columns are only processed when the UPDATE statement is nested in the FROM clause of a SELECT statement or a SELECT INTO statement. There must be at least one assignment clause that specifies a *column-name* that is not an INCLUDE column. The null value is returned for an included column that is not set by using an explicit SET clause.

For a positioned update, allowable column names can be further restricted to those in a certain list. This list appears in the FOR UPDATE clause of the SELECT statement for the associated cursor. The clause can be omitted by using the conditions that are described in "Positioned updates of columns" on page 333.

A view column that is derived from the same column as another column of the view can be updated, but both columns cannot be updated in the same UPDATE statement.

expression
Indicates the new value of the column. The *expression* is any expression of the type described in "Expressions" on page 240. It must not include an aggregate function.

If *expression* is a host variable, the host variable can include an indicator variable. When extended indicator variables are enabled, an expression must not be more complex than a reference to a single host variable if the indicator is set to an extended indicator value of default (-5) or unassigned (-7). In addition, a CAST specification can be used if either of the following is true:

- The target column is defined as nullable.
- the target column is defined as NOT NULL with a non-null default, the source of the CAST specification is a single host variable, and the data attributes (data type, length, precision, and scale) of the host variable are the same as the result of the cast specification.

A *column-name* in an expression must identify a column of the table or view. For each row that is updated, the value of the column in the expression is the value of the column in the row before the row is updated.

DEFAULT
Specifies that the default value is used based on how the corresponding column is defined in the table. The value that is assigned depends on how the column is defined.

- If the column is defined using the **IDENTITY** clause, the column is generated by the DB2 system.
- If the column is defined as a row change timestamp column, the column value is generated by the DB2 system.
- If the column is defined using the **WITH DEFAULT** clause, the value is set to the default that is defined for the column.
- If the column is defined without specifying the **WITH DEFAULT** clause, the **GENERATED** clause, or the **NOT NULL** clause, the value is NULL.
- If the column is specified in the **INCLUDE** column list, the column value is set to null.

A ROWID column must not be set to the **DEFAULT** keyword.

An identity column or a row change timestamp column that is defined as GENERATED ALWAYS can be set only to the **DEFAULT** keyword.

If the column is defined using the **NOT NULL** clause and the **GENERATED** clause is not used, or the **WITH DEFAULT** clause is not used, the **DEFAULT** keyword cannot be specified for that column.

NULL
Specifies the null value as the new value of the column. Specify **NULL** only for nullable columns.

row-fullselect
Specifies a fullselect that returns a single row. The column values are assigned to each of the corresponding *column-names*. If the fullselect returns no rows, the null value is assigned to each column; an error occurs if any column to be updated is not nullable. An error also occurs if there is more than one row in the result.

For a positioned update, if the table or view that is the object of the UPDATE statement is used in the fullselect, a column from the instance of the table or view in the fullselect cannot be the same as *column-name*, a column being updated.

If the fullselect refers to columns to be updated, the value of such a column in the fullselect is the value of the column in the row before the row is updated.

UNPACK-function-invocation
Specifies an invocation of the UNPACK built-in function. The number of fields that are returned by the UNPACK function invocation must be the same as the number of *column-names*.

WHERE
Specifies the rows to be updated. You can omit the clause, give a search condition, or specify a cursor. If you omit the clause, all rows of the table or view are updated.

search-condition
Specifies any search condition described in Chapter 2, "Language elements," on page 53. Each *column-name* in the search condition, other than in a subquery, must identify a column of the table or view.

The *search-condition* is applied to each row of the table or view and the updated rows are those for which the result of the *search-condition* is true. If the unique key or primary key is a parent key, the constraints are effectively checked at the end of the operation.

If the search condition contains a subquery, the subquery can be thought of as being executed each time the search condition is applied to a row, and the results used in applying the search condition. In actuality, a subquery with no correlated references is executed just once, whereas it is possible that a subquery with a correlated reference must be executed once for each row.

WHERE CURRENT OF *cursor-name*
Identifies the cursor to be used in the update operation. *cursor-name* must identify a declared cursor as explained in the description of the DECLARE CURSOR statement in "DECLARE CURSOR" on page 1535. If the UPDATE statement is embedded in a program, the DECLARE CURSOR statement must include *select-statement* rather than *statement-name*.

The object of the UPDATE statement must also be identified in the FROM clause of the SELECT statement of the cursor. The columns to be updated can be identified in the FOR UPDATE clause of that SELECT statement though they do not have to be identified. If the columns are not specified, the columns that can be updated include all the updatable columns of the table or view that is identified in the first FROM clause of the fullselect.

The result table of the cursor must not be read-only. For an explanation of read-only result tables, see Read-only cursors. Note that the object of the UPDATE statement must not be identified as the object of the subquery in the WHERE clause of the SELECT statement of the cursor.

When the UPDATE statement is executed, the cursor must be open and positioned on a row or rowset of the result table.

- If the cursor is positioned on a single row, that row is the one updated.
- If the cursor is positioned on a rowset, all rows corresponding to the rows of the current rowset are updated.

A positioned UPDATE must not be specified for a cursor that references a view on which an instead of update trigger is defined, even if the view is an updatable view.

FOR ROW *n* **OF ROWSET**
Specifies which row of the current rowset is to be updated. The corresponding row of the rowset is updated, and the cursor remains positioned on the current rowset.

host-variable or *integer-constant* is assigned to an integral value *k*. If *host-variable* is specified, it must be an exact numeric type with scale zero, must not include an indicator variable, and *k* must be in the range of 1 to 32767.

The cursor must be positioned on a rowset, and the specified value must be a valid value for the set of rows most recently retrieved for the cursor. If the specified row cannot be updated, an error is returned. It is possible that the specified row is within the bounds of the rowset most recently requested, but the current rowset contains less than the number of rows that were implicitly or explicitly requested when that rowset was established.

If this clause is not specified, the cursor position determines the rows that will be affected. If the cursor is positioned on a single row, that row is the one updated. In the case where the most recent FETCH statement returned multiple rows of data (but not as a rowset), this position would be on the last row of data that was returned. If the cursor is positioned on a rowset, all rows corresponding to the current rowset are updated. The cursor position remains unchanged.

It is possible for another application process to update a row in the base table of the SELECT statement so that the specified row of the cursor no longer has a corresponding row in the base table. An attempt to update such a row results in an error.

isolation-clause
Specifies the isolation level used when locating the rows to be updated by the statement.

WITH
Introduces the isolation level, which may be one of the following:
RR Repeatable read
RS Read stability
CS Cursor stability

The default isolation level of the statement is the isolation level of the package or plan in which the statement is bound, with the package isolation taking precedence over the plan isolation. When a package isolation is not specified, the plan isolation is the default.

SKIP LOCKED DATA
Specifies that rows are skipped when incompatible locks are held on the row by other transactions. These rows can belong to any accessed table that is specified in the statement. **SKIP LOCKED DATA** can be used only when isolation CS or RS is in effect and applies only to row level or page level locks.

SKIP LOCKED DATA can be specified only in the searched UPDATE statement (or the searched update operation of a MERGE statement). **SKIP LOCKED DATA** is ignored if it is specified when the isolation level that is in effect is repeatable read (**WITH RR**) or uncommitted read (**WITH UR**). The default isolation level of the statement depends on the isolation level of the package or plan with which the statement is bound, with the package isolation taking precedence over the plan isolation. When a package isolation is not specified, the plan isolation is the default.

QUERYNO *integer*
Specifies the number to be used for this SQL statement in EXPLAIN output and trace records. The number is used for the QUERYNO column of the plan table for the rows that contain information about this SQL statement. This number is also used in the QUERYNO column of the SYSIBM.SYSSTMT and SYSIBM.SYSPACKSTMT catalog tables.

If the clause is omitted, the number associated with the SQL statement is the statement number assigned during precompilation. Thus, if the application program is changed and then precompiled, that statement number might change.

Using the **QUERYNO** clause to assign unique numbers to the SQL statements in a program is helpful:
- For simplifying the use of optimization hints for access path selection
- For correlating SQL statement text with EXPLAIN output in the plan table

For information on using optimization hints, such as enabling the system for optimization hints and setting valid hint values, and for information on accessing the plan table, see *DB2 Performance Monitoring and Tuning Guide*.

Notes

Update rules:
Update values must satisfy the following rules. If they do not, or if other errors occur during the execution of the UPDATE statement, no rows are updated and the position of the cursors are not changed.
- *Assignment*. Update values are assigned to columns using the assignment rules described in Chapter 2, “Language elements,” on page 53.
- *Validity*. Updates must obey the following rules. If they do not, or if any other errors occur during the execution of the UPDATE statement, no rows are updated.
  - *Fullselects:* The row-fullselect and expressions that contain a *scalar-fullselect* must return no more than one row.
  - *Unique constraints and unique indexes:* If the identified table (or base table of the identified view) has any unique indexes or unique constraints, each row that is updated in the table must conform to the limitations that are imposed by those indexes and constraints.

All uniqueness checks are effectively made at the end of the statement. In the case of a multi-row update, this validation occurs after all the rows are updated.

- *Check constraints:* If the identified table (or base table of the identified view) has any check constraints, each check constraint must be true or unknown for each row that is updated in the table.

  All checks constraints are effectively validated at the end of the statement. In the case of a multi-row update, this validation occurs after all the rows are updated.

- *Views and the WITH CHECK OPTION*. For views defined with WITH CHECK OPTION, an updated row must conform to the definition of the view. If the view you name is dependent on other views whose definitions include WITH CHECK OPTION, the updated rows must also conform to the definitions of those views. For an explanation of the rules governing this situation, see "CREATE VIEW" on page 1527.

  For views that are not defined with WITH CHECK OPTION, you can change the rows so that they no longer conform to the definition of the view. Such rows are updated in the base table of the view and no longer appear in the view.

- *Field and validation procedures*. The updated rows must conform to any constraints imposed by any field or validation procedures on the identified table (or on the base table of the identified view).

- *Referential constraints*. The value of the parent key in a parent row must not be changed. If the update value produces a foreign key that is nonnull, the foreign key must be equal to some value of the parent key of the parent table of the relationship.

  All referential constraints are effectively checked at the end of the statement. In the case of a multi-row update, this validation occurs after all the rows are updated.

- *Indexes with VARBINARY columns*. If the identified table has an index on a VARBINARY column or a column that is a distinct type that is based on VARBINARY data type, that index column cannot specify the DESC attribute. To use the SQL data change operation on the identified table, either drop the index or alter the data type of the column to BINARY and then rebuild the index.

- *Triggers*. An UPDATE statement might cause triggers to activate. A trigger might cause other statements to be executed or raise error conditions based on the update values.If an UPDATE statement for a view causes an instead of trigger to activate, validity, referential integrity, and check constraints are checked against the data changes that are performed in the trigger and not against the view that causes the trigger to activate or its underlying base tables.

Number of rows updated:

Normally, after an UPDATE statement completes execution, the value of SQLERRD(3) in the SQLCA is the number of rows updated. (For a complete description of the SQLCA, including exceptions to the preceding sentence, see "SQL communication area (SQLCA)" on page 2069.)

Nesting user-defined functions or stored procedures:

An UPDATE statement can implicitly or explicitly refer to user-defined functions or stored procedures. This is known as *nesting* of SQL statements. A user-defined function or stored procedure that is nested within the UPDATE must not access the table being updated.

Locking:

Unless appropriate locks already exist, one or more exclusive locks are acquired by the execution of a successful update operation. Until a commit or rollback operation releases the locks, only the application process that performed the insert can access the updated row. If LOBs are not updated, application processes that are running with uncommitted read can also access the updated row. The locks can also prevent other application processes from performing operations on the table. However, application processes that are running with uncommitted read can access locked pages and rows.

Locks are not acquired on declared temporary tables.

Datetime representation when using datetime registers:

As explained under Datetime special registers, when two or more datetime registers are implicitly or explicitly specified in a single SQL statement, they represent the same point in time. This is also true when multiple rows are updated.

Rules for positioned UPDATE with a SENSITIVE STATIC scrollable cursor:

When a SENSITIVE STATIC scrollable cursor has been declared, the following rules apply:

- *Update attempt of delete holes.* If, with a positioned update against a SENSITIVE STATIC scrollable cursor, an attempt is made to update a row that has been identified as a delete hole, an error occurs.
- *Update operations.* Positioned update operations with SENSITIVE STATIC scrollable cursors perform as follows:
  1. The SELECT list items in the target row of the base table of the cursor are compared with the values in the corresponding row of the result table (that is, the result table must still agree with the base table). If the values are not identical, then the update operation is rejected, and an error occurs. The operation may be attempted again after a successful FETCH SENSITIVE has occurred for the target row.
  2. The WHERE clause of the SELECT statement is re-evaluated to determine whether the current values in the base table still satisfy the search criteria. The values in the SELECT list are compared to determine that these values have not changed. If the WHERE clause evaluates as true, and the values in the SELECT have not changed, the update operation is allowed to proceed. Otherwise, the update operation is rejected, an error occurs, and an *update hole* appears in the cursor.
- *Update of update holes.* Update holes are not permanent. It is possible for another process, or a searched update in the same process, to update an update hole row so that it is no longer an update hole. Update holes become visible with a FETCH SENSITIVE for positioned updates and positioned deletes.
- *Result table.* After the base table is updated, the row is re-evaluated and updated in the temporary result table. At this time, it is possible that the positioned update changed the data such that the row does not qualify the search condition, in which case the row is marked as an update hole for subsequent FETCH operations.

Updating rows in a table with multilevel security:

When you update rows in a table with multilevel security, DB2 compares the security label of the user (the primary authorization ID) to the security label of the row. The update proceeds according to the following rules:

- If the security label of the user and the security label of the row are equivalent, the row is updated and the value of the security label is determined by whether the user has write-down privilege:
  - If the user has write-down privilege or write-down control is not enabled, the user can set the security label of the row to any valid security label. The value that is specified for the security label column must be assignable to a column that is defined as CHAR(8) FOR SBCS DATA NOT NULL.
  - If the user does not have write-down privilege and write-down control is enabled, the security label of the row is set to the value of the security label of the user.
- If the security label of the user dominates the security label of the row, the result of the UPDATE statement is determined by whether the user has write-down privilege:
  - If the user has write-down privilege or write-down control is not enabled, the row is updated and the user can set the security label of the row to any valid security label.
  - If the user does not have write-down privilege and write-down control is enabled, the row is not updated.
- If the security label of the row dominates the security label of the user, the row is not updated.

Updating rows in a table for which row or column access control is enforced:
When an UPDATE statement is issued for a table for which row or column access control is enforced, the rules specified in the enabled row permissions or column masks determine whether the row can be updated. Typically those rules are based on the authorization ID or role of the process. The following describes how enabled row permissions and column masks are used during UPDATE:

- Row permissions are used to identify the set of rows to be updated.

  When multiple enabled row permissions are defined for a table, a row access control search condition is derived by application of the logical OR operator to the search condition in each enabled permission. This row access control search condition is applied to the table to determine which rows are accessible to the authorization ID or role of the UPDATE statement. If the WHERE clause is specified in the UPDATE statement, the user-specified predicates are applied on the accessible rows to determine the rows to be updated. If there is no WHERE clause, the accessible rows are the rows to be updated.

  Column masks are not applicable in this step.

  If the table is not enforced by row access control, the WHERE clause determines the rows to be updated, otherwise all rows in the table are to be updated.
- If there are rows to be updated, the following rules determine whether those rows can be updated:
  - For every column to be updated, the new value of the column must not be affected by enabled column masks whose columns are referenced when deriving the new value.

    When a column is referenced while deriving the values of a new row, if the column has an enabled column mask, the masked value is used to derive the new values. If the object table is also column access control activated, the column mask applied to derive the new values must ensure the evaluation of the access control rules defined in the

column mask resolves the column to itself, not to a constant or an expression. If the column mask does not mask the column to itself, the new value cannot be used for update and an error is returned at run time.

- If the rows are updatable, and there is a BEFORE UPDATE trigger for the table, the trigger is activated.

  Within the trigger actions, the new values for update might be modified in transition variables. When the final values are returned from the trigger, the new values are used for the update.
- The rows that are to be updated must conform to the enabled row permissions:

  For each row that is to be updated, the old values are replaced with the new values that were specified in the UPDATE statement. A row that conforms to the enabled row permissions is a row that, if updated, can be retrieved using the derived row access control search condition.
- If the rows are updatable, and there is an AFTER UPDATE trigger for the table, the trigger is activated.

The above rules are not applicable to the included columns. The included columns are subject to the rules for the select list because they are not the columns of the object table of the UPDATE statement.

Extended indicator variable usage:
When extended indicator variables are enabled, negative indicator values that are outside the range of -1 through -7 must not be specified, and the default and unassigned extended indicator values must not appear in contexts in which they are not supported.

Extended indicator variables:
Assigning an extended indicator value of unassigned has the effect of leaving the target column set to its current value, as if it had not been specified in the statement. Assigning an extended indicator value of default assigns the default value to the column.

If a target column is not updatable (for example, an identity column that is defined as GENERATED ALWAYS), it must be assigned the extended indicator value of unassigned unless the **OVERRIDING USER VALUE** clause is specified.

The UPDATE statement must not assign all target columns to the extended indicator value of unassigned.

Extended indicator variables and update triggers:
If a target column has been assigned an extended indicator value of unassigned, the column is not considered to have been updated. The column is treated as if it had not been specified in the **OF** *column-name* list of any update trigger that is defined on the target table.

Extended indicator variables and deferred error checks:
When extended indicator variables are enabled, validation that would otherwise be done in statement preparation (to recognize an update of a non-updatable column) is deferred until statement execution. If statement validation fails, an error is returned when the statement is run, not when the statement is prepared.

Considerations for a generated column:
A generated column that is defined as **GENERATED ALWAYS** should not be specified as the target of an assignment clause unless the value that is

to be assigned is specified with the DEFAULT keyword or an extended indicator that specifies that a default value is to be assigned. The user can specify the OVERRIDING USER VALUE clause to indicate that any user-specified value should be ignored and that DB2 should assign the default value.

Considerations for a system-period temporal table:

When a row of a system-period temporal table is updated, DB2 updates the values of the *row-begin* and *transaction-start-ID* columns as follows:

- A row-begin column is assigned a value that is generated by using the time-of-day clock during the execution of the first data change statement in the transaction that requires a value to be assigned to a row-begin column or transaction-start-ID column in a table. This also occurs when a row in a system-period temporal table is deleted. DB2 ensures the uniqueness of the generated values for a row-begin column across transactions. If multiple rows are updated within a single SQL transaction, the values for the row-begin column are the same for all the rows and are unique from the values that are generated for the column for another transaction.
- A transaction-start-ID column is assigned a unique timestamp value per transaction or the null value. The null value is assigned to the transaction-start-ID column if the column is nullable. Otherwise, the value is generated by using the time-of-day clock during execution of the first data change statement in the transaction that requires a value to be assigned to a row-begin column or transaction-start-ID column in a table. This also occurs when a row in a system-period temporal table is deleted. If multiple rows are updated within a single SQL transaction, the values for the transaction-start-ID column are the same for all the rows and are unique from the values that are generated for the column for another transaction.

If the UPDATE statement has a search condition that contains a correlated subquery that references historical rows (explicitly referencing the name of the history table or implicitly referenced through the use of a period specification in the FROM clause), the old version of the updated rows that are inserted as historical rows (into the history table) are potentially visible to update operations for the rows that are subsequently processed for the statement.

Considerations for a history table:

When a row of a system-period temporal table is updated, a historical copy of the row is inserted into the corresponding history table and the end timestamp of the historical row is captured in the form of a system determined value that corresponds to the time of the data change operation. DB2 generates the value by using the time-of-day clock during the execution of the first data change statement in the transaction that requires a value to be assigned to a row-begin or transaction-start-ID column in a table. This also occurs when a row in a system-period temporal table is deleted. DB2 ensures uniqueness of the generated values for an end column in a history table across transactions. If a conflicting transaction is updating the same row in the system-period temporal table and the row that is to be inserted into the associated history table would have an end timestamp value that is greater than the begin timestamp value, an error is returned.

Considerations for an application-period temporal table:

An UPDATE statement for an application-period temporal table that

contains a **FOR PORTION OF BUSINESS_TIME** clause indicates between which two points in time that the specified updates are effective. When **FOR PORTION OF BUSINESS_TIME** is specified, and the period value for a row that is specified by the values of the begin column and end column for the BUSINESS_TIME period is only partially contained in the period that is specified from *value1* up to *value2*, the row is updated and one or two rows are automatically inserted to represent the portion of the row that is not changed. New values are generated for each generated column in an application-period temporal table for each row that is automatically inserted as a result of an update operation on the table. If a generated column is defined as part of a unique or primary key, parent key in a referential constraint, or unique index, it is possible that an automatic insert can violate a constraint or index in which case an error is returned.

Effect of the CURRENT TEMPORAL SYSTEM_TIME and CURRENT TEMPORAL BUSINESS_TIME special registers

If the CURRENT TEMPORAL SYSTEM_TIME special register is set to a non-null value, the underlying target of the UPDATE statement cannot be a system-period temporal table. This restriction applies regardless of whether the system-period temporal table is directly or indirectly referenced.

If the CURRENT TEMPORAL BUSINESS_TIME special register is set to a non-null value, the UPDATE statement is affected if the following conditions are also true:

- An application-period temporal table is the target of the UPDATE statement.
- The BUSTIMESENSITIVE bind option is set to YES.

In this situation, DB2 implicitly adds the following additional predicates to the statement:

```
    bt_begin <= CURRENT TEMPORAL BUSINESS_TIME
AND bt_end > CURRENT TEMPORAL BUSINESS_TIME
```

In the preceding code, bt_begin and bt_end are the begin and end columns of the BUSINESS_TIME period of the target table of the UPDATE statement.

Archive-enabled tables:

A reference to an archive-enabled table as the target of the UPDATE statement does not affect rows in the associated archive table.

A data change statement must not reference an archive-enabled table when a system-period temporal table or application-period temporal table is also referenced.

Other SQL statements in the same unit of work:

The following statements cannot follow an UPDATE statement in the same unit of work:

- An ALTER TABLE statement that changes the data type of a column (ALTER COLUMN SET DATA TYPE)
- An ALTER INDEX statement that changes the padding attribute of an index with varying-length columns (PADDED to NOT PADDED or vice versa)

Using UPDATE to reset AREO* status on a table:

An UPDATE statement will reset the AREO* state of a table if all conditions are true:

- The statement is a searched UPDATE statement. An UPDATE statement within a SELECT statement will not reset the AREO* state.
- The expression in the SET clause is not a *scalar-fullselect* or *row-fullselect*
- The update operation is against a table in a universal table space
- The table does not have row access control activated
- The SKIP LOCKED DATA clause is not specified
- The WHERE clause is not specified
- A resource unavailable condition is not encountered.

  No error or warning SQLCODE is returned if a resource unavailable condition is encountered. Only a resource unavailable console message will be displayed.

A DISPLAY DATABASE command can be used to determine if AREO* is reset.

Examples

Example 1: Change employee 000190's telephone number to 3565 in DSN8B10.EMP.

```
UPDATE DSN8B10.EMP
  SET PHONENO='3565'
  WHERE EMPNO='000190';
```

Example 2: Give each member of department D11 a 100-dollar raise.

```
UPDATE DSN8B10.EMP
  SET SALARY = SALARY + 100
  WHERE WORKDEPT = 'D11';
```

Example 3: Employee 000250 is going on a leave of absence. Set the employee's pay values (SALARY, BONUS, and COMMISSION) to null.

```
UPDATE DSN8B10.EMP
  SET SALARY = NULL, BONUS = NULL, COMM = NULL
  WHERE EMPNO='000250';
```

Alternatively, the statement could also be written as follows:

```
UPDATE DSN8B10.EMP
  SET (SALARY, BONUS, COMM) = (NULL, NULL, NULL)
  WHERE EMPNO='000250';
```

Example 4: Assume that a column named PROJSIZE has been added to DSN8B10.EMP. The column records the number of projects for which the employee's department has responsibility. For each employee in department E21, update PROJSIZE with the number of projects for which the department is responsible.

```
UPDATE DSN8B10.EMP
  SET PROJSIZE = (SELECT COUNT(*)
                  FROM DSN8B10.PROJ
                  WHERE DEPTNO = 'E21')
  WHERE WORKDEPT = 'E21';
```

Example 5: Double the salary of the employee represented by the row on which the cursor C1 is positioned.

```
EXEC SQL UPDATE DSN8B10.EMP
  SET SALARY = 2 * SALARY
  WHERE CURRENT OF C1;
```

Example 6: Assume that employee table EMP1 was created with the following statement:

```
CREATE TABLE EMP1
  (EMP_ROWID    ROWID GENERATED ALWAYS,
   EMPNO        CHAR(6),
   NAME         CHAR(30),
   SALARY       DECIMAL(9,2),
   PICTURE      BLOB(250K),
   RESUME       CLOB(32K));
```

Assume that host variable *HV_EMP_ROWID* contains the value of the ROWID column for employee with employee number '350000'. Using that ROWID value to identify the employee and user-defined function UPDATE_RESUME, increase the employee's salary by $1000 and update that employee's resume.

```
EXEC SQL UPDATE EMP1
  SET SALARY = SALARY + 1000,
      RESUME = UPDATE_RESUME(:HV_RESUME)
  WHERE EMP_ROWID = :HV_EMP_ROWID;
```

Example 7: In employee table X, give each employee whose salary is below average a salary increase of 10%.

```
EXEC SQL UPDATE EMP X
  SET SALARY = 1.10 * SALARY
  WHERE SALARY < (SELECT AVG(SALARY) FROM EMP Y
  WHERE X.JOBCODE = Y.JOBCODE);
```

Example 8: Raise the salary of the employees in department 'E11' whose salary is below average to the average salary.

```
EXEC SQL UPDATE EMP T1
  SET SALARY = (SELECT AVG(T2.SALARY) FROM EMP T2)
  WHERE WORKDEPT = 'E11' AND
        SALARY < (SELECT AVG(T3.SALARY) FROM EMP T3);
```

Example 9: Give the employees in department 'E11' a bonus equal to 10% of their salary.

```
EXEC SQL
  DECLARE C1 CURSOR FOR
    SELECT BONUS
    FROM DSN8710.EMP
    WHERE WORKDEPT = 'E12'
    FOR UPDATE OF BONUS;
EXEC SQL
  UPDATE DSN8710.EMP
    SET BONUS = ( SELECT .10 * SALARY FROM DSN8710.EMP Y
                  WHERE EMPNO = Y.EMPNO )
    WHERE CURRENT OF C1;
```

Example 10: Assuming that cursor CS1 is positioned on a rowset consisting of 10 rows in table T1, update all 10 rows in the rowset.

```
EXEC SQL UPDATE T1 SET C1 = 5 WHERE CURRENT OF CS1;
```

Example 11: Assuming that cursor CS1 is positioned on a rowset consisting of 10 rows in table T1, update the fourth row of the rowset.

```
short ind1, ind2;

int n, updt_value;

stmt = 'UPDATE T1 SET C1 = ? WHERE CURRENT OF CS1 FOR ROW ? OF ROWSET'

ind1 = 0;
```

```
ind2 = 0;

n = 4;

updt_value = 5;

...

strcpy(my_sqlda.sqldaid,"SQLDA");

my_sqlda.sqln = 2;

my_sqlda.sqld = 2;

my_sqlda.sqlvar[0].sqltype = 497;
my_sqlda.sqlvar[0].sqllen = 4;
my_sqlda.sqlvar[0].sqldata = (int *) &updt_value;
my_sqlda.sqlvar[0].sqlind = (short *) &ind1;

my_sqlda.sqlvar[1].sqltype = 497;
my_sqlda.sqlvar[1].sqllen = 4;
my_sqlda.sqlvar[1].sqldata = (int *) &n;
my_sqlda.sqlvar[1].sqlind = (short *) &ind2;

EXEC SQL PREPARE S1 FROM :stmt;

EXEC SQL EXECUTE S1 USING DESCRIPTOR :my_sqlda;
```

Example 12: Assume that table POLICY exists and that it is defined with a single period, BUSINESS_TIME. The table contains a row where column BK has a value of 'P138' and column CLIENT has a value of 'C882', and column TYPE has a value of 'PPO'. Update the portion of the row beginning from the current date to set the TYPE column to 'HMO':

```
UPDATE POLICY
 FOR PORTION OF BUSINESS_TIME
  FROM CURRENT DATE TO DATE '9999-12-31'
  SET TYPE='HMO'
  WHERE BK='P138', CLIENT='C882';
```

After the UPDATE statement is processed, the table contains 2 rows in place of the original row. One row represents a value of 'PPO' for the TYPE column (the value before the update) and the other row represents a value of 'HMO' for the TYPE column (that began with the UPDATE statement).

Example 13: Suppose that the INTARRAY and CHARARRAY array types, the INTA, CHARA, and SI variables, and the T1 table are defined as follows:

```
CREATE TYPE INTARRAY AS INTEGER ARRAY [6];
CREATE TYPE CHARARRAY AS CHAR(20) ARRAY [7];
DECLARE INTA AS INTARRAY;
DECLARE CHARA AS CHARARRAY;
CREATE VARIABLE SI INT;
CREATE TABLE T1 (COL1 CHAR(7), COL2 INT);
```

Assign values to CHARA, INTA, and SI.

```
SET CHARA = ARRAY [ 'a', 'b', 'c' ];
SET INTA = ARRAY [ 1, 2, 3, 4, 5 ];
SET SI = 1;
```

Insert a row into table T1, and then update the row values using values from the CHARA and INTA arrays, which are indexed by the value of variable SI.

```
INSERT INTO T1 VALUES ('abc', 10);
UPDATE T1
 SET COL1 = CHARA[SI],
 COL2 = INTA[SI];
```

In the table row, COL1 now contains 'a', and COL2 contains 1.

Set the value of column COL2 for all rows to the cardinality of array INTA.

```
UPDATE T1
 SET COL2 = CARDINALITY(INTA);
```

In the table row, COL2 now contains 5.

VALUES

The VALUES statement provides a method for invoking a user-defined function from a trigger. Transition variables and transition tables can be passed to the user-defined function.

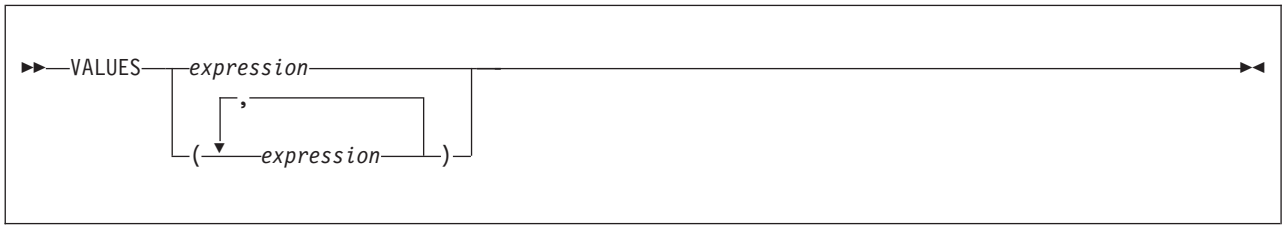
Invocation

This statement can only be used in the triggered action of a trigger.

Authorization

Authorization is required for any expressions that are used in the statement. For more information, see "Expressions" on page 240.

Syntax



Description

VALUES

Specifies one or more expressions. If more than one expression is specified, the expressions must be enclosed within parentheses.

expression

Any expression of the type described in "Expressions" on page 240. The expression must not contain a host variable.

The expressions are evaluated, but the resulting values are discarded and are not assigned to any output variables.

If a user-defined function is specified as part of an expression, the user-defined function is invoked. If a negative SQLCODE is returned when the function is invoked, DB2 stops executing the trigger and rolls back any triggered actions that were performed.

Example

Example: Create an after trigger EMPISRT1 that invokes user-defined function NEWEMP when the trigger is activated. An insert operation on table EMP activates the trigger. Pass transition variables for the new employee number, last name, and first name to the user-defined function.

```
CREATE TRIGGER EMPISRT1
   AFTER INSERT ON EMP
   REFERENCING NEW AS N
   FOR EACH ROW
   MODE DB2SQL
   BEGIN ATOMIC
      VALUES(NEWEMP(N.EMPNO, N.LASTNAME, N.FIRSTNAME));
   END
```

VALUES INTO

The VALUES INTO statement assigns one or more values to variables.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

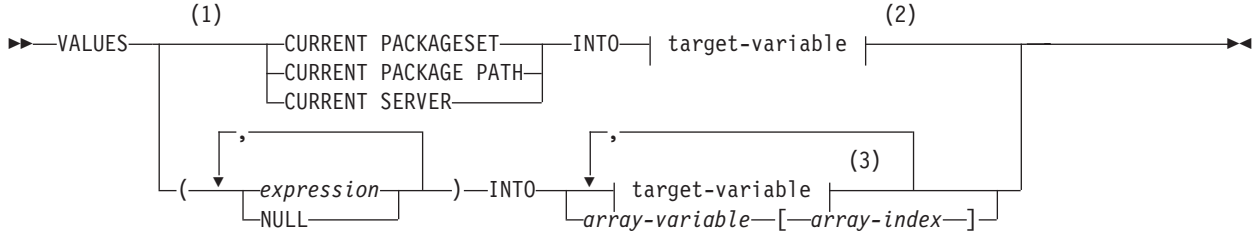
Authorization

The privileges that are held by the authorization ID of the statement must include at least one of the following privileges or authorities:

- The SELECT privilege on every table and view identified in the statement
- Ownership of every table and view identified in the statement
- READ and WRITE privileges on any global variables that are identified in the statement
- Ownership of any global variables that are identified in the statement
- DBADM authority for the database (tables only)
- DATAACCESS authority
- SYSADM authority
- SYSCTRL authority (catalog tables only)

Authorization is required for any expressions that are used in the statement. For more information, see "Expressions" on page 240.

Syntax



target-variable:

Notes:

1 These special registers can be referenced only as a source value in this form of the syntax for this statement.

2 *target-variable* must not be an array type in this context.

3 The number of source value specifications (*expression*, NULL, or DEFAULT) on the right side of the equal sign must match the number of target specifications on the left side of the statement.

Description

VALUES

Introduces a single row that consists of one or more columns. If more than one value is specified, the list of values must be enclosed within parentheses.

expression

The expression is any expression of the type described in "Expressions" on page 240, except that it cannot contain a reference to the CURRENT PACKAGESET, CURRENT PACKAGE PATH, or CURRENT SERVER special registers. The expression must not include a column name.

NULL

The null value. **NULL** can only be specified for host variables that have an associated indicator variable.

INTO *target-variable* **or** *array-variable*[*array-index*]

Identifies one or more targets for the assignment of output values. The number of targets in the INTO clause must equal the number of values that are to be assigned. The first value in the result row is assigned to the first target in the list, the second value to the second target, and so on. A target variable must not be specified more than once in the INTO clause. Each assignment to a target is made in sequence through the list, according to the rules described in "Assignment and comparison" on page 121.

The value 'W' is assigned to the SQLWARN3 field of the SQLCA if the number of targets is less than the number of result column values.

If an error occurs on any assignment, the value is not assigned to the target, and no more values are assigned to the specified targets. Any values that have already been assigned remain assigned.

global-variable-name
Identifies the global variable that is the assignment target.

host-variable-name
Identifies the host variable that is the assignment target. For LOB output values, the target can be a regular host variable (if it is large enough), a LOB locator variable, or a LOB file reference variable.

SQL-parameter-name
Identifies the parameter that is the assignment target.

SQL-variable-name
Identifies the SQL variable that is the assignment target. SQL variables must be declared before they are used.

array-variable **[***array-index***]**
Specifies an array element that is the target of the assignment.

An array element must not be specified as the target for an assignment if *common-table-expression* is also specified in the statement.

[*array-index***]**
An expression that specifies which element in the array is the target of the assignment.

For an ordinary array, the array index expression must be castable to INTEGER, and must not be the null value. The index value must be between 1 and the maximum cardinality that is defined for the array.

For an associative array, the array index expression must be castable to the index data type of the associative array, and must not be the null value.

array-index must not be:

- An expression that references the CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP special register
- A nondeterministic function
- A function that is defined with EXTERNAL ACTION
- A function that is defined with MODIFIES SQL DATA
- A sequence expression

Notes

Assignment to targets:
The *n*th target identified by the INTO clause or described in the SQLDA corresponds to the *n*th column of the result table of the cursor. The data type of target must be compatible with its corresponding value. If the value is numeric, the target must have the capacity to represent the whole part of the value. For a datetime value, the target must be a character string variable of a minimum length as defined in "String representations of datetime values" on page 101. When the target is a host variable, if the value is null, an indicator variable must be specified.

Assignments are made in sequence through the list. Each assignment to a target is made according to the rules described in Chapter 2, "Language elements," on page 53. If the number of targets is less than the number of

values in the row, the SQLWARN3 field of the SQLCA is set to 'W'. There is no warning if there are more targets than the number of result columns. If the target is a host variable and the value is null, an indicator variable must be provided. If an assignment error occurs, the value is not assigned to the target and no more values are assigned to targets. Any values that have already been assigned to targets remain assigned. However, if LOB values are involved, there is a possibility that the corresponding target was modified, but the variable's contents are unpredictable.

If more than one assignment is included in the same assignment statement, all expressions are evaluated before the assignments are performed. For example, a reference to a variable in an expression always uses the value of the variable prior to any assignment in the assignment statement.

Normally, you use LOB locators to assign and retrieve data from LOB columns. However, because of compatibility rules, you can also use LOB locators to assign data to targets with other data types. For more information on using locators, see Saving storage when manipulating LOBs by using LOB locators (DB2 Application programming and SQL).

Default encoding scheme:

The default encoding scheme for the data is the value in the bind option ENCODING, which is the option for application encoding. If this statement is used with functions such as LENGTH or SUBSTRING that are operating on LOB locators, and the LOB data that is specifies by the locator is in a different encoding scheme from the ENCODING bind option, LOB materialization and character conversion occur. To avoid LOB materialization and character conversion, select the LOB data from the SYSIBM.SYSDUMMYA, SYSIBM.SYSDUMMYE, or SYSIBM.SYSDUMMYU sample table.

Examples

Example 1: Assign the value of the CURRENT PATH special register to host variable *HV1*.

```
EXEC SQL VALUES(CURRENT PATH)
         INTO :HV1;
```

Example 2: Assign the value of the CURRENT MEMBER special register to host variable *MEM*.

```
EXEC SQL VALUES(CURRENT MEMBER)
         INTO :MEM;
```

Example 3: Assume that LOB locator *LOB1* is associated with a CLOB value. Assign a portion of the CLOB value to host variable *DETAILS* using the LOB locator.

```
EXEC SQL VALUES (SUBSTR(:LOB1,1,35))
         INTO :DETAILS;
```

If the LOB data that is specified by the LOB locator *LOB1* is in a different encoding scheme from the value of the ENCODING bind option, and you want to avoid LOB materialization and character conversion, use the following statement instead of the VALUES INTO statement:

```
EXEC SQL SELECT SUBSTR(:LOB1,1,35)
         INTO :DETAILS
         FROM SYSIBM.SYSDUMMYU;
```

Example 4: Using a VALUES INTO statement, retrieve the value of INTVAR1 into an element in array MYINTARRAY1, which is indexed by the value of the expression INTCOL2+MYINTVAR+1.

```
VALUES INTVAR1 INTO MYINTARRAY1[INTCOL2+MYINTVAR+1];
```

WHENEVER

The WHENEVER statement specifies the host language statement to be executed when a specified exception condition occurs.

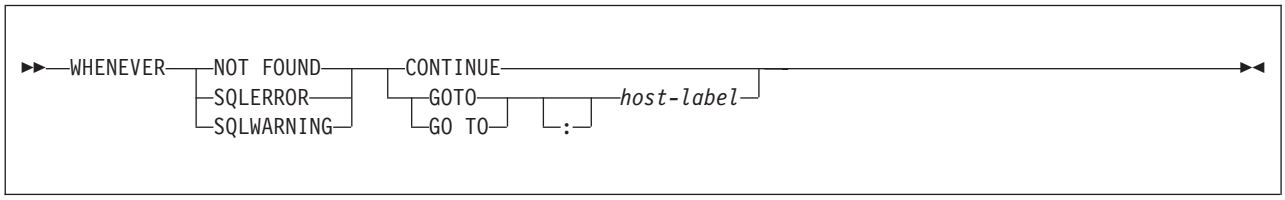
Invocation

This statement can only be embedded in an application program. It is not an executable statement. It must not be specified in Java or REXX.

Authorization

None required.

Syntax



Description

The **NOT FOUND**, **SQLERROR**, or **SQLWARNING** clause is used to identify the type of exception condition.

NOT FOUND
Identifies any condition that results in an SQLCODE of +100 (equivalently, an SQLSTATE code of '02000').

SQLERROR
Identifies any condition that results in a negative SQLCODE.

SQLWARNING
Identifies any condition that results in a warning condition (SQLWARN0 is W), or that results in a positive SQLCODE other than +100.

The **CONTINUE** or **GO TO** clause specifies the next statement to be executed when the identified type of exception condition exists.

CONTINUE
Specifies the next sequential statement of the source program.

GOTO or GO TO *host-label*
Specifies the statement identified by *host-label*. For *host-label*, substitute a single token, optionally preceded by a colon. The form of the token depends on the host language. In COBOL, for example, it can be *section-name* or an unqualified *paragraph-name*.

Notes

There are three types of WHENEVER statements:

- WHENEVER NOT FOUND
- WHENEVER SQLERROR
- WHENEVER SQLWARNING

Every executable SQL statement in an application program is within the scope of one implicit or explicit WHENEVER statement of each type. The scope of a WHENEVER statement is related to the listing sequence of the statements in the application program, not their execution sequence.

An SQL statement is within the scope of the last WHENEVER statement of each type that is specified before that SQL statement in the source program. If a WHENEVER statement of some type is not specified before an SQL statement, that SQL statement is within the scope of an implicit WHENEVER statement of that type in which **CONTINUE** is specified. If a WHENEVER statement is specified in a Fortran subprogram, its scope is that subprogram, not the source program.

The GET DIAGNOSTICS statement can be used to provide additional information.

Examples

The following statements can be embedded in a COBOL program.

Example 1: Go to the label HANDLER for any statement that produces an error.

```
EXEC SQL WHENEVER SQLERROR GOTO HANDLER END-EXEC.
```

Example 2: Continue processing for any statement that produces a warning.

```
EXEC SQL WHENEVER SQLWARNING CONTINUE END-EXEC.
```

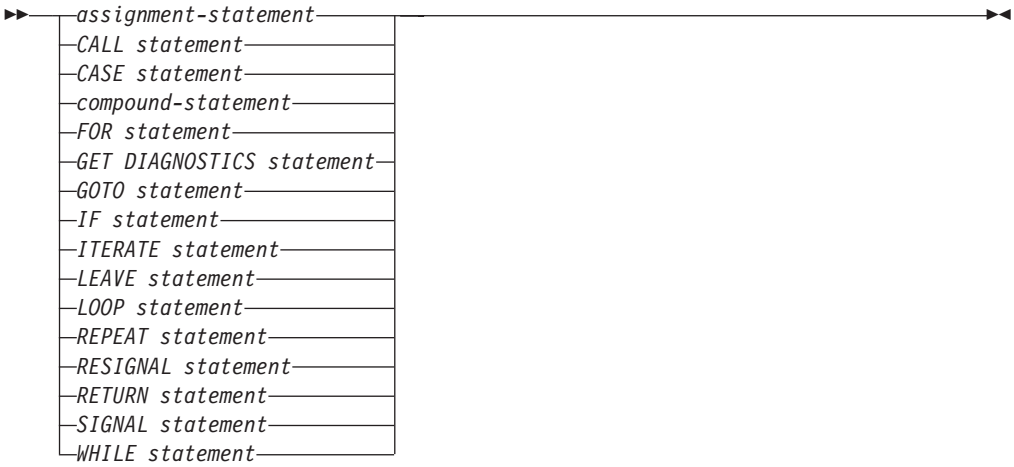
Example 3: Go to the label ENDDATA for any statement that does not return.

```
EXEC SQL WHENEVER NOT FOUND GO TO ENDDATA END-EXEC.
```

Chapter 6. SQL control statements for SQL routines

SQL control statements for SQL routines can be used in SQL functions and native SQL procedures. *SQL control statements* provide the capability to control the logic flow, declare and set variables, and handle warnings and exceptions. Some SQL control statements include other nested SQL statements.

SQL-control-statement:



Control statements are supported in SQL functions and SQL procedures.

- SQL functions are created by specifying LANGUAGE SQL and an SQL routine body in a CREATE FUNCTION statement. An SQL function can be changed. A new SQL routine body can be specified in an ALTER FUNCTION statement.
- SQL procedures are created by specifying LANGUAGE SQL and an SQL routine body in a CREATE PROCEDURE statement. An SQL procedure can be changed. A new SQL routine body can be specified in an ALTER PROCEDURE statement.

The SQL routine body must be a single SQL statement, which might be an SQL control statement.

The SQL routine body is the executable part of the function or procedure and is transformed by DB2 into a program.

The remainder of the topics about SQL control statements for SQL routines contain information about references to SQL parameters and variables, SQL condition names, SQL cursor names, labels, and reference information for the use of the statements that constitute the SQL routine body.

The two common elements that are used in describing specific SQL control statements are:

- SQL control statements as described above
- "SQL-procedure-statement" on page 1968

References to SQL parameters and SQL variables

SQL parameters and SQL variables can be referenced anywhere in the statement where an expression or a variable can be specified. Host variables cannot be specified in SQL routines. SQL parameters can be referenced anywhere in the routine and can be qualified with the routine name. SQL variables can be referenced anywhere in the compound statement in which they are declared, including any statement that is directly or indirectly nested within that compound statement.

If the compound statement where the variable is declared has a label, references to the variable name can be qualified with that label.

All SQL parameters and SQL variables are considered nullable. The name of an SQL parameter or an SQL variable in an SQL routine can be the same as the name of a column in a table or view that is referenced in the routine. The name of an SQL variable can also be the same as the name of another SQL variable that is declared in the same routine. This can occur when the two SQL variables are declared in different *compound-statements*. The *compound-statement* that contains the declaration of an SQL variable determines the scope of that variable. See "compound-statement" on page 1977 for additional information.

Names that are the same should be explicitly qualified. Qualifying a name clearly indicates whether the name refers to a column, an SQL variable, or an SQL parameter. If the name is not qualified or is qualified but is still ambiguous, the following rules describe whether the name refers to a column or to an SQL variable or an SQL parameter in an SQL routine:

- The name is checked to see if it is the name of a column of any existing table or a view that is specified in the SQL routine body at the current server. If the name is found as a column name, but the privilege set that is used to issue the CREATE PROCEDURE or ALTER PROCEDURE statement does not have the proper authority to access the table or view, the VALIDATE option that is in effect for the procedure determines what happens:
  - If VALIDATE BIND is in effect, an error is returned.
  - If VALIDATE RUN is in effect, the name is assumed to be a column name. If the privilege set that is used to issue the CREATE statement does not have the proper authority to access the table or view at run time, an error is returned.
- If the referenced tables or views do not exist at the current server, the name will be checked first as an SQL variable name in the compound statement and then as an SQL parameter name. The variable can be declared within the *compound-statement* that contains the reference, or within a compound statement in which that compound statement is nested. If two SQL variables are within the same scope and have the same name, the SQL variable that is declared in the innermost compound statement is used.

  If the name is not found as an SQL parameter or SQL variable, the VALIDATE option that is in effect for the procedure determines what happens:
  - If VALIDATE BIND is in effect, an error is returned.
  - If VALIDATE RUN is in effect, the name is assumed to a be column name. If a column does not exist with that name at run time, an error is returned.

The name of an SQL variable or an SQL parameter in an SQL routine can be the name of an identifier that is used in certain SQL statements. If the name is not qualified, the following rules describe whether the name refers to an identifier, an SQL variable, or and SQL parameter:

- In the SET CURRENT PACKAGE PATH, SET PATH and SET SCHEMA statements, the name is checked as an SQL variable name or an SQL parameter name. If an SQL variable or SQL parameter with that name is not found, the name is assumed to be an identifier.
- In the ASSOCIATE LOCATORS, CONNECT, RELEASE (connection), and SET CONNECTION statements, the name is used as an identifier.

References to SQL condition names

A condition name can only be referenced within the compound statement in which it is declared, including any compound statements that are nested within that compound statement. When there is a reference to a condition name, the condition that is declared in the innermost compound statement is the condition that is used.

The name of an SQL condition can be the same as the name of another SQL condition that is declared in the same routine. This can occur when the two SQL conditions are declared in different *compound-statements*. The *compound-statement* that contains the declaration of an SQL condition name determines the scope of that condition name. A condition name must be unique within the compound statement in which it is declared, excluding any declarations in compound statements that are nested within that compound statement. A condition name can only be referenced within the compound statement in which it is declared, including any compound statements that are nested within that compound statement. When there is a reference to a condition name, the condition that is declared in the innermost compound statement is the condition that is used. See “compound-statement” on page 1977 for additional information.

References to SQL cursor names

A cursor name can only be referenced within the compound statement in which it is declared, including any compound statements that are nested within that compound statement.

The name of an SQL cursor can not be the same as the name of another SQL cursor that is declared on the same routine. The *compound-statement* that contains the declaration of an SQL cursor name determines the scope of that cursor name. A cursor name can only be referenced within the compound statement in which it is declared, including any compound statements that are nested within that compound statement. See “compound-statement” on page 1977 for additional information.

References to labels

Specifying a label for an SQL procedure statement defines that label and determines the scope of that label. A label name can only be referenced within the compound statement in which it is defined, including a reference from any statement that is directly or indirectly nested within that compound statement. The FOR statement is considered the same as a compound statement with respect to defining and referencing labels. A label can be specified as the target of a GOTO, LEAVE, or ITERATE statement, subject to the rules for the statement that references the label as a target.

Labels can be specified on most SQL procedure statements. If a label is specified on an SQL procedure statement, it must be unique from other labels within the same scope. A label must not be the same as any other label within the same compound statement, must not be the same as a label specified on the compound statement itself, and if the compound statement is nested within another compound statement, the label must not be the same as the label specified on any higher level compound statement. The label must not be the same as the name of the SQL procedure.

Nested compound statements and scope of names

Nested compound statements can be used within an SQL routine to define the scope of SQL variable declarations, cursors, condition names, and condition handlers.

In addition, labels have a defined scope in the context of nested compound statements. However, the rules for name spaces and how non-unique names can be referenced, differs depending on the type of name. The following table summarizes these differences:

Table 153. Scope and qualification of names within nested compound statements

Type of name	Name can be qualified	Name must be unique within	Name can be referenced within
SQL variable	Yes. The name can be qualified with the label of the compound statement in which the variable is declared	the compound statement in which it is declared, excluding any declarations in compound statements that are nested within that compound statement	The compound statement in which it is declared, including any compound statements that are nested within that compound statement. When multiple SQL variables are defined with the same name, a label can be used to explicitly refer to a specific variable that is not the most local in scope
condition name	No	the compound statement in which it is declared, excluding any declarations in compound statements that are nested within that compound statement	The compound statement in which it is declared, including any compound statements that are nested within that compound statement. Condition names can be used in the declaration of a condition handler, or in a SIGNAL or RESIGNAL statement. If multiple conditions are defined with the same name, there is no way to explicitly refer to the condition that is not the most local in scope.

Table 153. Scope and qualification of names within nested compound statements (continued)

Type of name	Name can be qualified	Name must be unique within	Name can be referenced within
cursor name	No	the routine	The compound statement in which it is declared, including any compound statements that are nested within that compound statement. If the cursor is defined as a result set cursor, the invoking application can access the result set.
label	No	the compound statement that defined the label, including any definitions in compound statements that are nested within that compound statement	The compound statement in which it is defined, including any compound statements that are nested within that compound statement. Use a label to qualify the name of an SQL variable or as the target of a GOTO, LEAVE, or ITERATE statement, subject to the rules for these statements.

SQL-procedure-statement

An SQL control statement can allow multiple SQL statements to be specified within the SQL control statement. These statements are defined as SQL procedure statements.

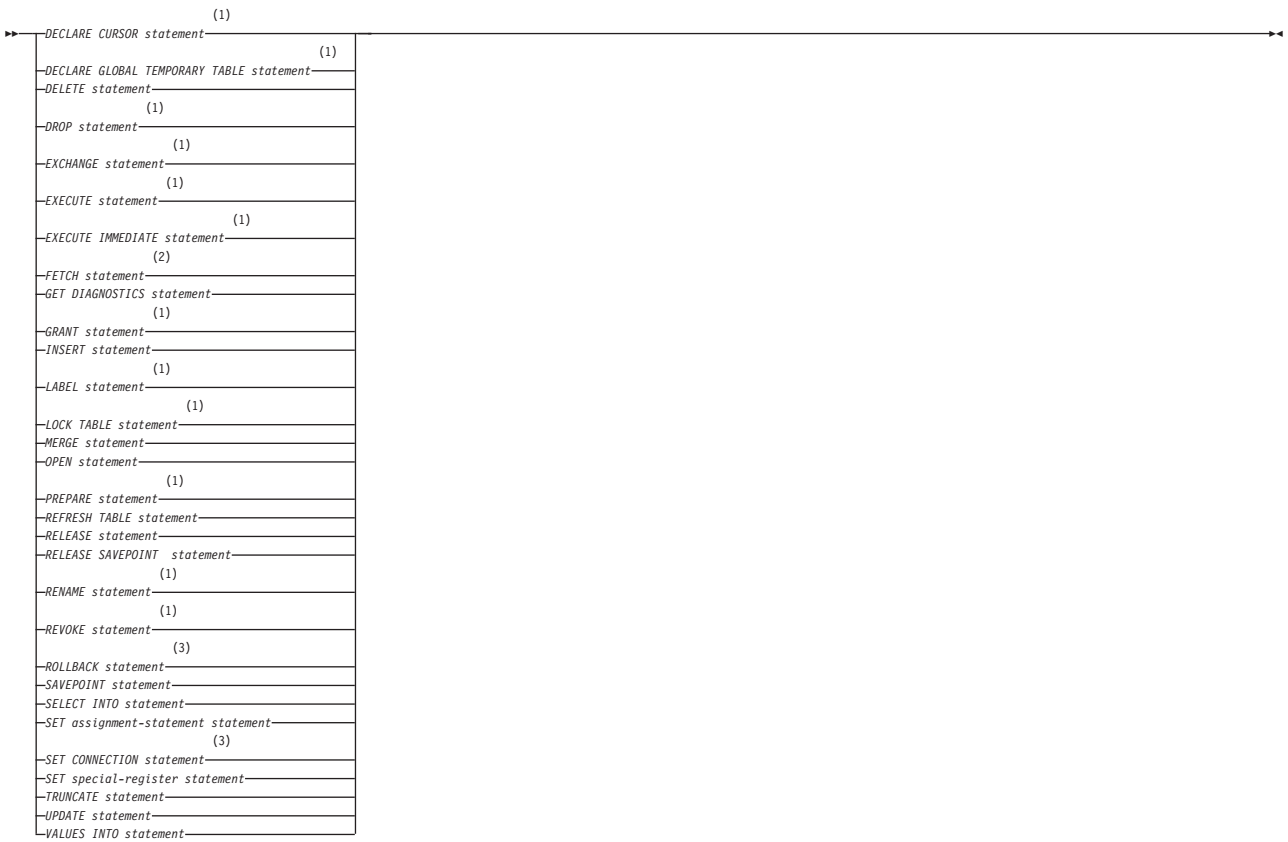
Syntax

Notes:

1 The statement is not allowed in an *SQL-routine-body* for an SQL function

2 An ALTER FUNCTION statement (SQL scalar) or an ALTER PROCEDURE statement (SQL - native) with an ADD VERSION or REPLACE clause is not allowed in an *SQL-routine-body*.

3 The COMMIT, ROLLBACK, CONNECT, and SET CONNECTION statements must only be specified within the body of an SQL procedure. The COMMIT statement and the ROLLBACK statement (without the TO SAVEPOINT clause) must not be issued in a routine body if the routine is in the calling chain of an SQL routine, or an external routine.



Notes:

1 The statement is not allowed in an *SQL-routine-body* for an SQL function

2 A FETCH statement must not specify a *fetch-orientation* clause, *multiple-row-fetch* clause, the WITH CONTINUE or the CURRENT CONTINUE clauses.

3 The COMMIT, ROLLBACK, CONNECT, and SET CONNECTION statements must only be specified within the body of an SQL procedure. The COMMIT statement and the ROLLBACK statement (without the TO SAVEPOINT clause) must not be issued in a routine body if the routine is in the calling chain of an SQL routine, or an external routine.

Description

SQL-control-statement
Specifies an SQL statement that provides the capability to control logic flow, declare and set variables, and handle warnings and exceptions, as defined in this section. Control statements are supported in SQL routines.

SQL-statement
Specifies an SQL statement. These statements are described in Chapter 5, "Statements," on page 833.

Notes

Comments: Comments can be included within the body of an SQL routine. In addition to the double-dash form of comments (--), a comment can begin with /* and end with */. The following rules apply to this form of comment:
- The beginning characters /* must be adjacent and on the same line.
- The ending characters */ must be adjacent and on the same line.
- Comments can be started wherever a space is valid.

- Comments can be continued to the next line.

Detecting and processing error and warning conditions: As an SQL statement is executed, DB2 stores information about the processing of the statement in a diagnostics area (including the SQLSTATE and SQLCODE), unless otherwise noted in the description of the SQL statement. A completion condition can indicate that the SQL statement completed successfully, completed with a warning condition, or completed with a not found condition. An exception condition indicates that the SQL statement was not successful.

A condition handler can be defined to execute when an exception condition, a warning condition, or a not found condition occurs in a compound statement. The declaration of a condition handler includes the code that is executed when the condition handler is activated. When a condition other than a successful completion occurs in the processing of *SQL-procedure-statement* and a condition handler that can handle the condition is within scope, one such condition handler will be activated to process the condition. See "compound-statement" on page 1977 for information about defining condition handlers. The code in the condition handler can check for a warning condition, a not found condition, or an exception condition and can take the appropriate action. Use one of the following methods at the beginning of the body of a condition handler to check the condition in the diagnostics area that caused the handler to be activated.

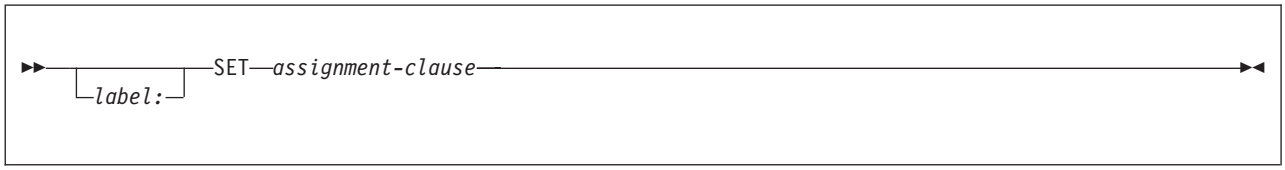
- Issue a GET DIAGNOSTICS statement to request the information from the diagnostics area. See "GET DIAGNOSTICS" on page 1679.
- Test the SQLSTATE and SQLCODE SQL variables.

If the condition is a warning and no handler exists for the condition, the previous two methods can be used outside of the body of a condition handler, if they are used immediately following the statement for which the condition is wanted. If the condition is an error and no handler exists for the condition, the routine terminates with the error condition.

assignment-statement

The assignment statement assigns a value to variables or array elements. The target value can be an SQL parameter or an SQL variable.

Syntax



Description

label

Specifies the label for *assignment-statement*. The label name cannot be the same as the name of the SQL routine, or another label within the same scope. For additional information, see "References to labels" on page 1965.

See "SET assignment-statement" on page 1875 for details.

Notes

Assignment rules: Assignment statements in SQL routines must conform to the SQL assignment rules. For example, the data type of the target and source must be compatible. See "Assignment and comparison" on page 121 for assignment rules.

When a string is assigned to a fixed-length variable and the length of the string is less than the length attribute of the target, the string is padded on the right with the necessary number of single-byte or double-byte blanks. When a string is assigned to a variable and the string is longer than the length attribute of the variable, the value is truncated and a warning is returned.

If truncation of the whole part of a number occurs on assignment to a numeric variable, the value is truncated and a warning is returned.

Assignments involving SQL parameters for SQL procedures: An IN parameter can appear on the left or right side in an assignment statement. When control returns to the caller, the original value of the IN parameter is retained. An OUT parameter can also appear on the left or right side in an assignment statement. If used without first being assigned a value, the value is undefined. When control returns to the caller, the last value that is assigned to an OUT parameter is returned to the caller. For an INOUT parameter, the first value of the parameter is determined by the caller, and the last value that is assigned to the parameter is returned to the caller.

Multiple assignments: If more than one assignment is included in the same assignment statement, all *expressions* are evaluated before the assignments are performed. Thus, references to an SQL variable or SQL parameter in an expression always use the value of the SQL variable or SQL parameter prior to any assignment in the assignment statement.

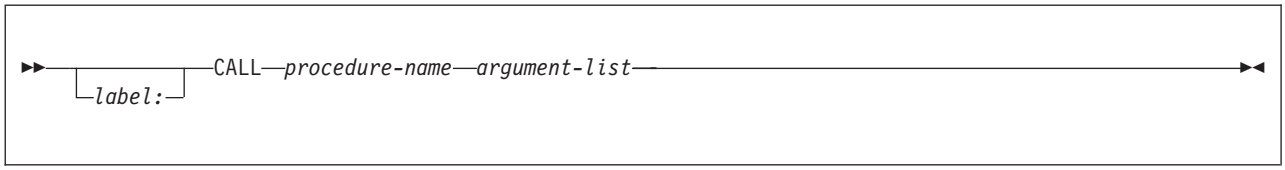
Considerations for SQLSTATE and SQLCODE SQL variables: Assignment to these variables is not prohibited. However, it is not recommended as assignment does not affect the diagnostic area or result in the activation of condition handlers.

Furthermore, processing an assignment to these SQL variables causes the specified values for the assignment to be overlayed with the SQL return codes returned from executing the statement that does the assignment.

CALL statement

The CALL statement invokes a stored procedure.

Syntax



argument-list:

Description

label
Specifies the label for the CALL statement. The label name cannot be the same as the name of the SQL routine or another label within the same scope. For additional information, see "References to labels" on page 1965.

procedure-name
Identifies the stored procedure to call. The procedure name must identify a stored procedure that exists at the current server.

argument-list
Identifies a list of values to be passed as parameters to the stored procedure. The *n*th value corresponds to the *n*th parameter in the procedure. The number of parameters must be the same as the number of parameters defined for the stored procedure. See "CALL" on page 1117 for more information.

Control is passed to the stored procedure according to the calling conventions for SQL routines. When execution of the stored procedure is complete, the value of each parameter of the stored procedure is assigned to the corresponding parameter of the CALL statement defined as OUT or INOUT.

SQL-variable-name
Specifies an SQL variable as an argument to the stored procedure. For an explanation of references to SQL variables, see "References to SQL parameters and SQL variables" on page 1964.

SQL-parameter-name
Specifies an SQL parameter as an argument to the stored procedure. For an explanation of references to SQL parameters, see "References to SQL parameters and SQL variables" on page 1964.

expression
The parameter is the result of the specified *expression*, which is evaluated

before the stored procedure is invoked. If *expression* is a single *SQL-parameter-name* or *SQL-variable-name*, the corresponding parameter of the procedure can be defined as IN, INOUT, or OUT. Otherwise, the corresponding parameter of the procedure must be defined as IN. If the result of the *expression* can be the null value, either the description of the procedure must allow for null parameters or the corresponding parameter of the stored procedure must be defined as OUT.

The following additional rules apply depending on how the corresponding parameter was defined in the CREATE PROCEDURE statement for the procedure:

- IN *expression* can contain references to multiple SQL parameters or variables. In addition to the rules stated in "Expressions" on page 240 for *expression*, *expression* cannot include a column name, an aggregate function, or a user-defined function that is sourced on an aggregate function.
- INOUT or OUT *expression* can only be a single SQL parameter or variable.

NULL
The parameter is a null value. The corresponding parameter of the procedure must be defined as IN and the description of the procedure must allow for null parameters.

Notes

See "CALL" on page 1117 for more information on the SQL CALL statement.

Examples

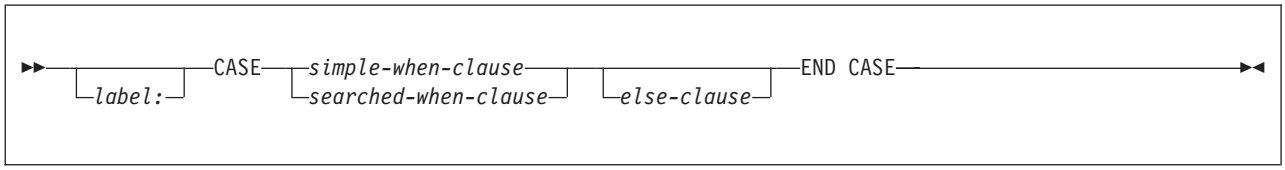
Call stored procedure proc1 and pass SQL variables as parameters.

```
CALL proc1(v_empno, v_salary)
```

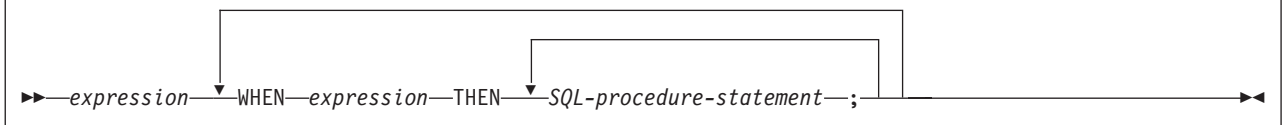
CASE statement

The CASE statement selects an execution path based on multiple conditions. A CASE statement operates in the same way as a CASE expression.

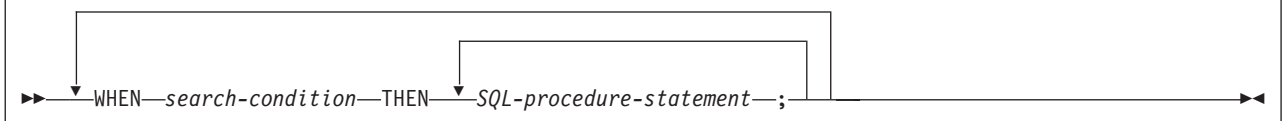
Syntax



simple-when-clause:



searched-when-clause:



else-clause:

Description

label
Specifies the label for the CASE statement. The label name cannot be the same as the name of the SQL routine or another label within the same scope. For additional information, see "References to labels" on page 1965.

CASE
Begins a *case-expression*.

simple-when-clause
The value of the *expression* prior to the first WHEN keyword is tested for equality with the value of the *expression* that follows each WHEN keyword. If the comparison is true, the statements in the associated THEN clause are

executed and processing of the CASE statement ends. If the result is unknown or false, processing continues to the next comparison. If the result does not match any of the comparisons, and an ELSE clause is present, the statements in the ELSE clause are executed.

searched-when-clause
The *search-condition* following the WHEN keyword is evaluated. If it evaluates to true, the statements in the associated THEN clause are executed and processing of the CASE statement ends. If it evaluates to false, or unknown, the next *search-condition* is evaluated. If no *search-condition* evaluates to true and an ELSE clause is present, the statements in the ELSE clause are executed.

When *searched-when-clause* is used, *search-condition* cannot contain a *fullselect*.

SQL-procedure-statement
Specifies a statement to execute. See"SQL-procedure-statement" on page 1968.

search-condition
Specifies a condition that is true, false, or unknown about a row or group of table data.

ELSE *SQL-procedure-statement*
If none of the conditions specified in the *simple-when-clause* or *searched-when-clause* are true, the statements specified in *SQL-procedure-statement* are executed.

If none of the conditions specified in the WHEN clauses are true and an ELSE is not specified, an error is issued when the statement executes, and the execution of the CASE statement is terminated.

END CASE
Ends a *case-statement*.

Examples

Example 1: Use a simple case statement WHEN clause to update column DEPTNAME in table DEPT, depending on the value of SQL variable *v_workdept*.

```
CASE v_workdept
 WHEN 'A00'
  THEN UPDATE DEPT SET
   DEPTNAME = 'DATA ACCESS 1';
 WHEN 'B01'
  THEN UPDATE DEPT SET
   DEPTNAME = 'DATA ACCESS 2';
 ELSE UPDATE DEPT SET
   DEPTNAME = 'DATA ACCESS 3';
END CASE
```

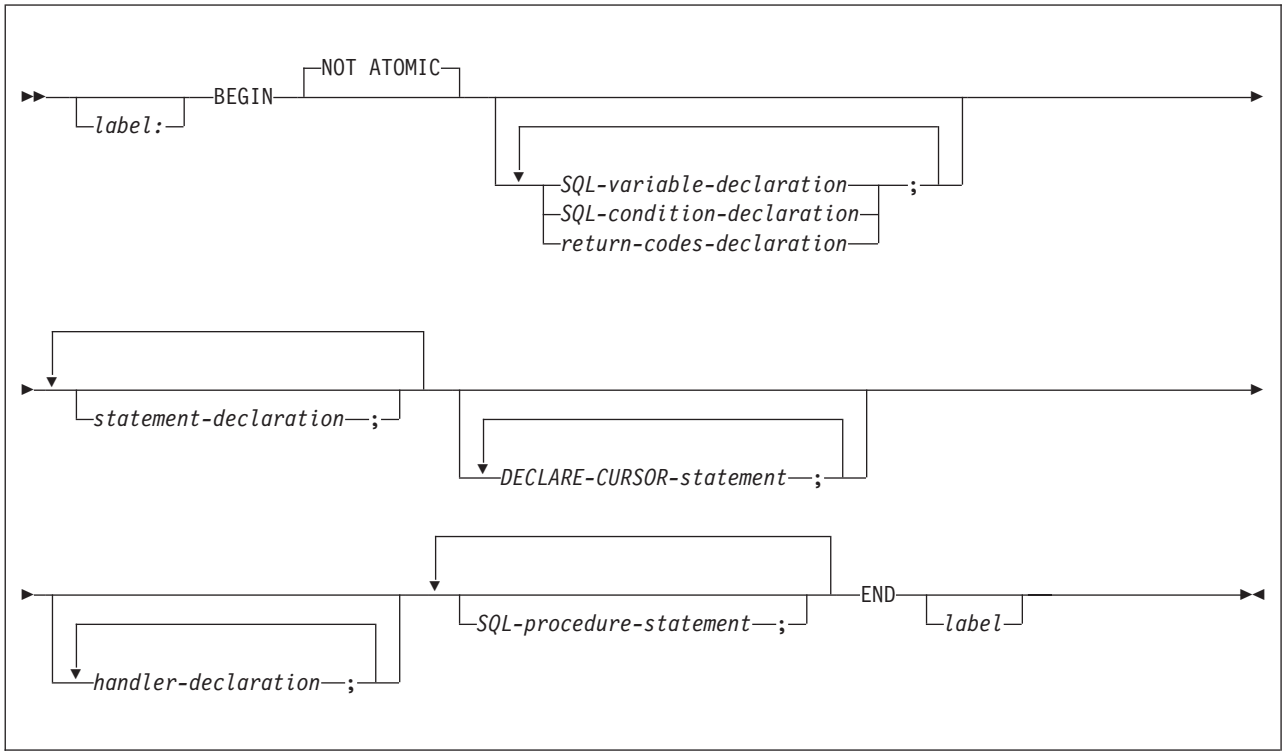
Example 2: Use a searched case statement WHEN clause to update column DEPTNAME in table DEPT, depending on the value of SQL variable *v_workdept*.

```
CASE
 WHEN v_workdept < 'B01'
  THEN UPDATE DEPT SET
   DEPTNAME = 'DATA ACCESS 1';
 WHEN v_workdept < 'C01'
  THEN UPDATE DEPT SET
   DEPTNAME = 'DATA ACCESS 2';
 ELSE UPDATE DEPT SET
   DEPTNAME = 'DATA ACCESS 3';
END CASE
```

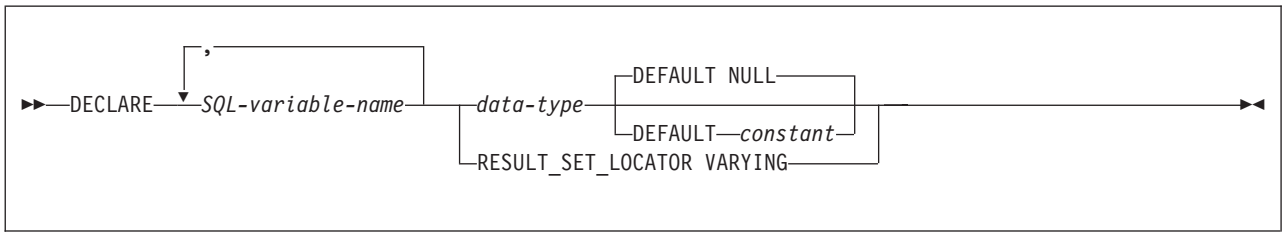
compound-statement

A compound statement groups other statements together in an SQL routine. A compound statement allows the declaration of SQL variables, cursors, and condition handlers.

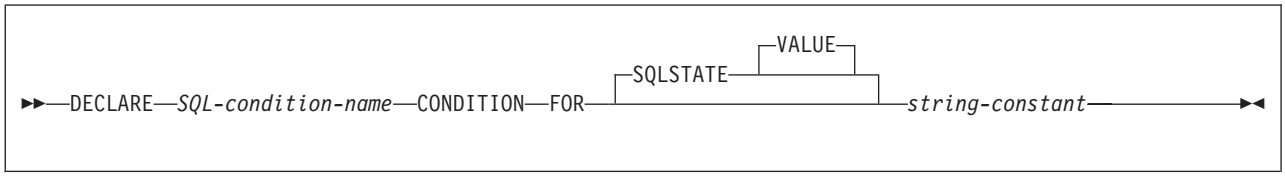
Syntax



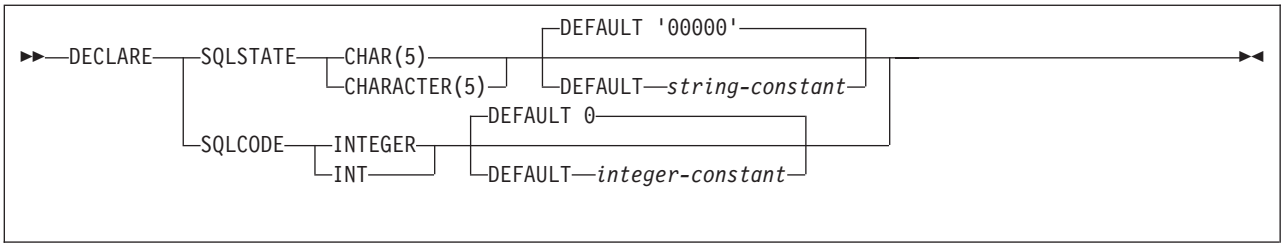
SQL-variable-declaration:



SQL-condition-declaration:

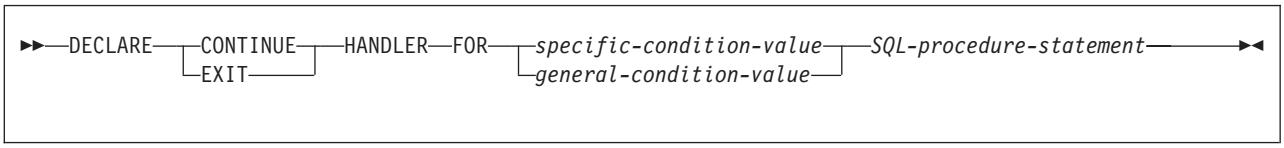


return-codes-declaration:

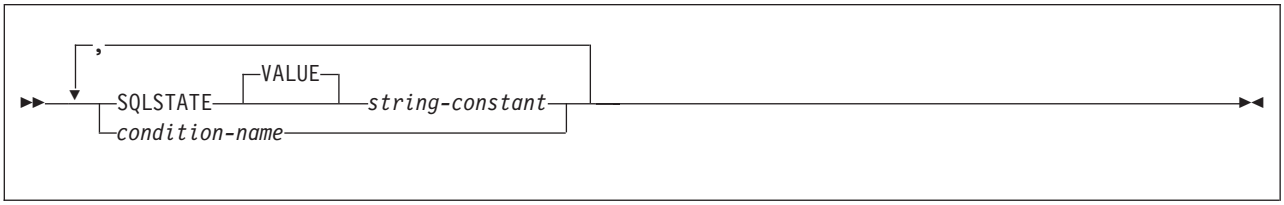


statement-declaration:

handler-declaration:



specific-condition-value:



general-condition-value:

Description

label

Specifies the label for the *compound-statement*. If the beginning label is specified, it can be used to qualify SQL variables declared in the compound statement and can also be specified as the target on a LEAVE statement. If the ending label is specified, it must be the same as beginning label. The label name cannot be the same as the routine name or another label within the same scope.

NOT ATOMIC
NOT ATOMIC indicates that an unhandled exception condition within the *compound-statement* does not cause the *compound-statement* to be rolled back.

SQL-variable-declaration
Declares a variable that is local to the compound statement.

SQL-variable-name
Defines the name of a variable. DB2 converts all SBCS SQL variable names that are not delimited to uppercase. *SQL-variable-name* must be unique within the compound statement in which is it declared, excluding any declarations in compound statements that are nested within that compound statement. *SQL-variable-name* must not be the same as a parameter name. See "References to SQL parameters and SQL variables" on page 1964 for information about how SQL variable names are resolved when there are columns with the same name as an SQL variable involved in a statement, or when multiple SQL variables exist with the same name in the routine body.

SQL-variable-name can only be referenced within the compound statement in which it is declared, including any compound statement that is nested within that compound statement. If the compound statement where the variable is declared has a label, references to the variable name can be qualified with that label. For example, an SQL variable V that is declared in a compound statement that is labeled C can be referenced as C.V.

data-type
Specifies the data type and length of the variable. SQL variables follow the same rules for default lengths and maximum lengths as SQL routine parameters. See "CREATE FUNCTION (SQL scalar)" on page 1224 and "CREATE PROCEDURE (SQL - native)" on page 1350 for descriptions of SQL data types and lengths.

DEFAULT *constant* **or NULL**
Defines the default for the SQL variable. The specified constant must represent a value that could be assigned to the variable in accordance with the rules of assignment as described in "Assignment and comparison" on page 121. The variable is initialized when the compound statement begins processing. If a default value is not specified, the SQL variable is initialized to NULL. Only DEFAULT NULL can be explicitly specified if *array-type-name* is specified.

RESULT_SET_LOCATOR VARYING
Specifies the data type for a result set locator variable.

SQL-condition-declaration
Declares a condition name and corresponding SQLSTATE value.

SQL-condition-name
Specifies the name of the condition. The condition name must be unique within the compound statement in which it is declared, excluding any declarations that are in compound statements that are nested within that compound statement. A condition name can only be referenced within the compound statement in which it is declared, including any compound statements that are nested within that compound statement.

FOR SQLSTATE *string-constant*
Specifies the SQLSTATE that is associated with the condition. The string must be specified as five characters enclosed in single quotes, and the SQLSTATE class (the first two characters) must not be '00'.

return-codes-declaration
Declares special variables named SQLSTATE and SQLCODE. These variables are automatically set to the SQLSTATE and SQLCODE values for the first condition in the diagnostics area after executing an SQL statement other than GET DIAGNOSTICS or an empty compound statement.

The SQLSTATE and SQLCODE SQL variables are only intended to be used as a means of obtaining the SQL return codes that resulted from processing the previous SQL statement other than GET DIAGNOSTICS. If there is any intention to use the SQLSTATE and SQLCODE values, save the values immediately to other SQL variables to avoid having the values replaced by the SQL return codes returned after executing the next SQL statement. If a handler is defined that handles an SQLSTATE, you can use an assignment statement to save that SQLSTATE (or the associated SQLCODE) value in another SQL variable, if the assignment is the first statement in the handler.

Assignment to these variables is not prohibited; however, it is not recommended. Assignment to these variables is ignored by condition handlers, and processing an assignment to these special variables causes the specified values for the assignment to be overlayed with the SQL return codes returned from executing the statement that does the assignment. The SQLSTATE and SQLCODE SQL variables cannot be set to NULL.

statement-declaration
Declares a list of one or more names that are local to the compound statement. A statement name cannot be the same as another statement name within the same compound statement.

DECLARE-CURSOR-statement
Declares a cursor in the procedure body. Each cursor must have a unique name within the routine. The cursor can only be referenced from within the compound statement in which it is declared, including any compound statements that are nested within that compound statement. Use an OPEN statement to open the cursor, a FETCH statement to read a row using the cursor, and a CLOSE statement to close the cursor. If the cursor is intended for use as a result set cursor:

- Specify WITH RETURN when the cursor is declared
- Create the procedure using the DYNAMIC RESULT SETS clause with a non-zero value
- Do not specify a CLOSE statement for the cursor in the compound statement

For additional information about declaring a cursor, see "DECLARE CURSOR" on page 1535.

handler-declaration
Specifies a condition handler, an *SQL-procedure-statement* to execute when an exception or completion condition occurs in the *compound-statement*. The *SQL-procedure-statement* executes when a condition handler receives control.

A condition handler declaration cannot reference the same condition value or SQLSTATE value more than one time. It cannot reference an SQLSTATE value and a condition name that represent the same SQLSTATE value.

When two or more condition handlers are declared in a compound statement, no two condition handler declarations can specify the same:

- general condition category
- specific condition, either as an SQLSTATE value or as a condition name that represents the same value

A condition handler is active for the set of *SQL-procedure-statements* that follow the condition handler declarations within the compound statement in which the condition handler is declared, including any nested compound statements.

CONTINUE
Specifies that after the condition handler is activated and completes successfully, control is returned to the SQL statement that follows the statement that raised the condition. However, if the condition is an error condition and it was encountered while evaluating a search condition, as in a CASE, FOR, IF, REPEAT or WHILE statement, control returns to the statement that follows the corresponding END CASE, END FOR, END IF, END REPEAT, or END WHILE.

EXIT
Specifies that after the condition handler is activated and completes successfully, control is returned to the end of the compound statement that declared the condition handler.

The conditions that can cause the handler to gain control are:

SQLSTATE *string-constant*
Specifies that the handler is invoked when the specific SQLSTATE occurs. The first two characters of the SQLSTATE value must not be '00'.

SQL-condition-name
Specifies that the handler is invoked when the specific SQLSTATE that is associated with the condition name occurs. The *SQL-condition-name* must be declared within the compound statement that contains the handler declarations, or within a compound statement in which that compound statement is nested.

SQLEXCEPTION
Specifies that the handler is invoked when an SQLEXCEPTION occurs. An SQLEXCEPTION is an SQLSTATE in which the class code is a value other than '00', '01', or '02'. For more information on SQLSTATE values, see *DB2 Codes*.

SQLWARNING
Specifies that the handler is invoked when an SQLWARNING occurs. An SQLWARNING is an SQLSTATE value with a class code of '01'.

NOT FOUND
Specifies that the handler is invoked when a NOT FOUND condition occurs. NOT FOUND corresponds to an SQLSTATE value with a class code of '02'.

Notes

Unlike host variables, SQL variables are not preceded by colons when they are used in SQL statements.

Nesting compound statements: Compound statements can be nested. Nested compound statements can be used to scope variable definitions, condition names, condition handlers, and cursors to a subset of the statements in a routine. This can simplify the processing that is done for each SQL routine statement. Nested compound statements enable the use of a compound statement within the declaration of a condition handler.

The scope of a cursor: The scope of a cursor name is the compound statement in which it is declared, including any compound statements that are nested within that compound statement. A cursor name can only be referenced within the compound statement in which it is declared, including any compound statements that are nested within that compound statement.

Considerations for statement-name: The scope of a *statement-name* that is declared in a compound statement is the compound statement and any nested compound statements (unless the same *statement-name* is declared in a nested compound statement). If a *statement-name* is used in a DECLARE CURSOR statement or a PREPARE statement and has not been declared in the compound statement where it is used or any outer compound statements in which it is nested, the statement-name is assumed to be declared globally for the routine.

Condition handlers: Condition handlers in SQL routines are similar to WHENEVER statements that are used in external SQL application programs. A condition handler can be defined to automatically get control when an exception, warning, or not found condition occurs. The body of a condition handler contains code that is executed when the condition handler is activated. A condition handler can be activated as the result of an exception, a warning, or a not found condition that is returned by DB2 for the processing of an SQL statement. Or the condition that activates the handler can be the result of a SIGNAL or RESIGNAL statement that is issued within the SQL routine body.

A condition handler is declared within a compound statement, and it is active for the set of *SQL-procedure-statements* that follow all of the condition handler declarations within the compound statement in which the condition handler is declared. For example, the scope of a condition handler declaration H is the list of *SQL-procedure-statements* that follow the condition handler declarations that are contained within the compound statement in which H appears. This means that the scope of H does not include the statements that are contained in the body of the condition handler H, implying that a condition handler cannot handle conditions that arise inside its own body. Similarly, for any two condition handlers H1 and H2 that are declared in the same compound statement, H1 will not handle conditions that arise in the body of H2, and H2 will not handle conditions that arise in the body of H1.

The declaration of a condition handler specifies the condition that activates it, the type of condition handler (CONTINUE or EXIT), and the handler action. The type of condition handler determines to where control is returned after the handler action successfully completes.

Condition handler activation: When a condition other than a successful completion occurs in the processing of *SQL-procedure-statement*, if a condition handler that could handle the condition is within scope, one such condition handler will be activated to process the condition.

In a routine with nested compound statements, condition handlers that could handle a specific condition might exist at several levels of the nested compound statements. The condition handler that is activated is a condition handler that is declared innermost to the scope in which the condition was encountered. If more than one condition handler at the nesting level could handle the condition, the condition handler that is activated is the most appropriate handler that is declared in that compound statement.

The most appropriate handler is the condition handler that most closely matches the SQLSTATE or the exception or completion condition. For a given compound statement, when both a specific handler for a condition and a general handler are declared that address the same condition, the specific handler takes precedence over the general handler.

For example, if the innermost compound statement declares a specific handler for SQLSTATE '22001', as well as a general handler for SQLEXCEPTION, the specific handler for SQLSTATE '22001' is the most appropriate handler when SQLSTATE '22001' is encountered. In this case, the specific handler is activated.

When a condition handler is activated, the condition handler action is executed. If the handler action completes successfully or with an unhandled warning, the diagnostics area is cleared, and the type of the condition handler (CONTINUE or EXIT handler) determines to where control is returned. Additionally, the SQLSTATE and SQLCODE SQL variables are cleared when a handler completes successfully or with an unhandled warning.

If the handler action does not complete successfully and an appropriate handler exists for the condition that is encountered in the handler action, that condition handler is activated. Otherwise, the condition that is encountered within the condition handler is unhandled.

Unhandled conditions: If a condition is encountered and an appropriate handler does not exist for that condition, the condition is unhandled.

- If the unhandled condition is an exception, the SQL routine that contains the failing statement is terminated with an unhandled exception condition.
- If the unhandled condition is a warning or is a not found condition, processing continues with the next statement. Note that the processing of the next SQL statement will cause information about the unhandled condition in the diagnostics area to be overwritten, and evidence of the unhandled condition will no longer exist.

Considerations for using SIGNAL and RESIGNAL statements with nested compound statements: If an *SQL-procedure-statement* that is specified in the condition handler is either a SIGNAL or RESIGNAL statement with an exception SQLSTATE, the compound statement terminates with the specified exception. This happens even when this condition handler or another condition handler in the same compound statement specifies CONTINUE, since these condition handlers are not in the scope of this exception. If a compound statement is nested in another compound statement, condition handlers in the higher level compound statement can handle the exception because those condition handlers are within the scope of the exception.

SQLSTATE and SQLCODE variables in SQL routines: To help debug your SQL routines, you might find it useful to check the SQLSTATE and SQLCODE value after executing a statement. An SQLCODE or SQLSTATE variable can be declared and subsequently referenced in an SQL routine. You could insert the value of the SQLCODE and SQLSTATE into a table at various points in the SQL routine, or return the SQLCODE and SQLSTATE values in a diagnostics string as an OUT parameter. To use the SQLCODE and SQLSTATE values, you must declare the following SQL variables in the SQL routine body:

```
DECLARE SQLCODE INTEGER DEFAULT 0;
DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
```

When you reference the SQLCODE or SQLSTATE variables in an SQL routine, DB2 sets the value of SQLCODE to 0 and SQLSTATE to '00000' for the subsequent statement. You can also use CONTINUE condition handlers to assign the value of the SQLSTATE and SQLCODE variables to variables in your SQL routine body. You can then use these SQL variables to control your routine logic, or pass the value back as an output parameter. In the following example, the SQL routine returns control to the statement following each SQL statement with the SQLCODE set in a SQL variable called RETCODE:

```
DECLARE SQLCODE INTEGER DEFAULT 0;
DECLARE retcode INTEGER DEFAULT 0;
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SET retcode = SQLCODE;
DECLARE CONTINUE HANDLER FOR SQLWARNING SET retcode = SQLCODE;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET retcode = SQLCODE;
```

The compound statement itself does not affect the SQLSTATE and SQLCODE SQL variables. However, SQL statements contained within the compound statement can affect the SQLSTATE and SQLCODE SQL variables. At the end of the compound statement, the SQLSTATE and SQLCODE SQL variables reflect the result of the last SQL statement executed within the compound statement that caused a change to the SQLSTATE and SQLCODE SQL variables. If the SQLSTATE and SQLCODE SQL variables were not changed within the compound statement, they contain the same values as when the compound statement was entered.

Null values in SQL parameters and SQL variables: If the value of an SQL parameter or SQL variable is null and it is used in an SQL statement that does not allow an indicator variable, an error is returned.

Effect on open cursors: At the end of the compound statement, all open cursors that are declared in that compound statement, except cursors that are used to return result sets, are closed.

Atomic processing of a compound statement: Atomic processing is not supported for a compound statement. If atomic behavior is needed for a block of code in a compound statement, set a savepoint before the nested compound statement is entered. This will allow changes to be undone with a ROLLBACK TO SAVEPOINT statement.

Examples

Example 1: Create a procedure body with a compound statement that performs the following actions:

- Declares SQL variables.
- Declares a cursor to return the salary of employees in a department determined by an IN parameter.
- Declares an EXIT handler for the condition NOT FOUND (end of file) which assigns the value 6666 to the OUT parameter *medianSalary*.
- Select the number of employees in the given department into the SQL variable *v_numRecords*.
- Fetch rows from the cursor in a WHILE loop until 50% + 1 of the employees have been retrieved.
- Return the median salary.

```
CREATE PROCEDURE DEPT_MEDIAN
   (IN deptNumber SMALLINT,
    OUT medianSalary DOUBLE)
   LANGUAGE SQL
```

```
BEGIN
  DECLARE v_numRecords INTEGER DEFAULT 1;
  DECLARE v_counter INTEGER DEFAULT 0;
  DECLARE c1 CURSOR FOR
     SELECT salary FROM staff
       WHERE DEPT = deptNumber
       ORDER BY salary;
  DECLARE EXIT HANDLER FOR NOT FOUND
     SET medianSalary = 6666;
     /* initialize OUT parameter */
     SET medianSalary = 0;
     SELECT COUNT(*) INTO v_numRecords FROM staff
        WHERE DEPT = deptNumber;
     OPEN c1;
     WHILE v_counter < (v_numRecords / 2 + 1) DO
        FETCH c1 INTO medianSalary;
        SET v_counter = v_counter + 1;
     END WHILE;
     CLOSE c1;
END
```

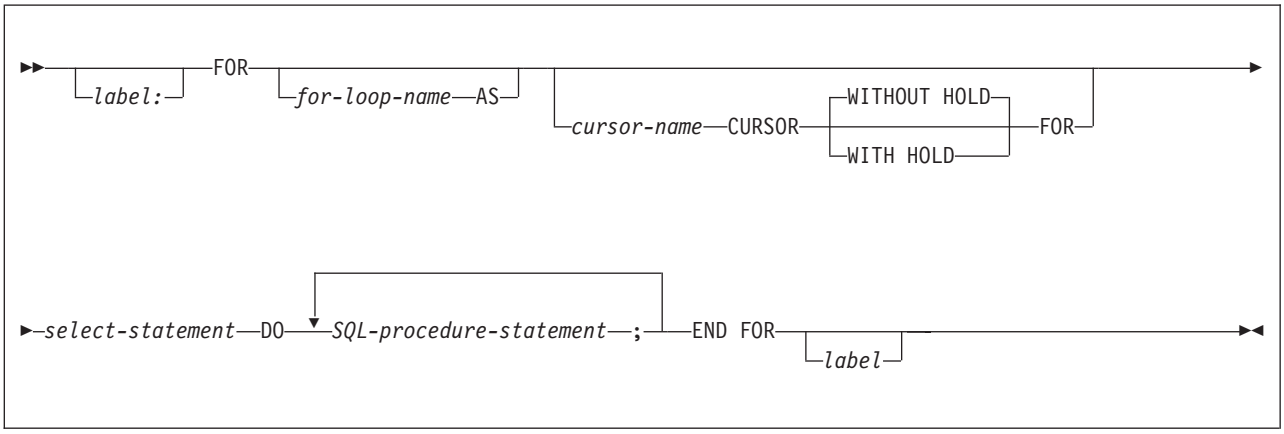
Example 2: Define an exit handler for any error, warning, or case of end of data. When this procedure is invoked and it returns to the caller, the value 45000 is returned for the output parameter:

```
CREATE PROCEDURE JMBLIB.PROCL(OUT MEDIANSALARY INT)
  LANGUAGE SQL
  BEGIN
   DECLARE CHAR1 CHAR;
   DECLARE C1 CURSOR FOR SELECT *
     FROM SYSIBM.SYSDUMMY1;
   DECLARE EXIT HANDLER FOR NOT FOUND,
        SQLEXCEPTION,
        SQLWARNING;
   OPEN C1;
   FETCH C1 INTO CHAR1;
   SET MEDIANSALARY = 45000;
   FETCH C1 INTO CHAR1;
   SET MEDIANSALARY = 50000;
  END
```

FOR statement

The FOR statement executes a statement for each row of a table. An implicit compound statement is generated to implement the FOR statement.

Syntax



Description

label
Specifies the label for the FOR statement. If the ending label is specified, it must be the same as the beginning label. The label name cannot be the same as the routine name or another label within the same scope. For more information, see "References to labels" on page 1965.

for-loop-name
Specifies the label for the implicit compound statement that is generated to implement the FOR statement. *for-loop-name* follows the rules for the label of a compound statement except that it cannot be used with an ITERATE, GOTO, or LEAVE statement within the FOR statement. *for-loop-name* must not be the same as any label within the same scope.

for-loop-name can be used to qualify generated SQL variables that correspond to the columns that are returned by *select-statement*.

cursor-name
Names a cursor that is generated to select rows from the result table of *select-statement*. If *cursor-name* is not specified, a unique cursor name is generated.

cursor-name cannot be referenced outside of the FOR statement and cannot be specified on an OPEN, FETCH, or CLOSE statement.

WITH HOLD or WITHOUT HOLD
Specifies whether the cursor should be prevented from being closed as a consequence of a commit operation.

WITHOUT HOLD
Specifies that the cursor is not prevented from being closed as a consequence of a commit operation. **WITHOUT HOLD** is the default.

WITH HOLD
Specifies that the cursor should not be closed as a consequence of a commit operation. A cursor that is declared using the **WITH HOLD** clause is implicitly closed at commit time only if the connection that is associated

with the cursor is ended during the commit operation. For more information, see "DECLARE CURSOR" on page 1535.

select-statement
Specifies the select statement of the cursor.

Each expression in the SELECT list must have a name. If an expression is not a simple column name, the AS clause must be used to name the expression. If the AS clause is specified, that name is used for the generated SQL variable that corresponds to the column returned by *select-statement*. The names for all of the generated SQL variables must be unique.

The SELECT list must not include an untyped array value.

SQL-procedure-statement
Specifies the SQL statements to be executed for each row of the table. The SQL statements must not include an OPEN, FETCH, or CLOSE statement that specifies the cursor name of the FOR statement.

Notes

FOR statement rules: The FOR statement executes one or multiple statements for each row in a table. The cursor is defined by specifying a SELECT list that describes the columns and rows selected. The statements within the FOR statement are executed for each row selected.

The SELECT list must consist of unique column names and the table that is specified in the FROM clause of *select-statement* must exist when the routine is created.

Handler warning: Handlers can be used to handle errors that might occur on the open of the cursor or fetch of a row using the cursor in the FOR statement. Handlers defined to handle these open or fetch conditions should not be CONTINUE handlers as they might cause the FOR statement to loop indefinitely.

Examples

In the following example, the FOR statement is used to specify a cursor that selects three columns from the employee table. For every row selected, SQL variable *fullname* is set to the last name followed by a comma, the first name, a blank, and the middle initial. Each value for *fullname* is inserted into table TNAMES.

```
BEGIN
  DECLARE fullname CHAR(40);
  FOR v1 AS
      c1 CURSOR FOR
   SELECT firstname, midinit, lastname FROM employee
    DO
      SET fullname =
          lastname CONCAT ', '
                   CONCAT firstname
                   CONCAT ' '
                   CONCAT midinit;
      INSERT INTO TNAMES VALUES ( fullname );
  END FOR;
END;
```

GET DIAGNOSTICS statement

The GET DIAGNOSTICS statement obtains information about the previous SQL statement that was executed.

See "GET DIAGNOSTICS" on page 1679.

When you need to specify a variable in a GET DIAGNOSTICS statement that is used within an SQL routine, you would use either *SQL-variable-name* or *SQL-parameter-name*. In an embedded GET DIAGNOSTICS statement, you would use a *host-variable*. You can replace the instances of *host-variable* in the description of "GET DIAGNOSTICS" on page 1679 with *SQL-variable-name* or *SQL-parameter-name*.

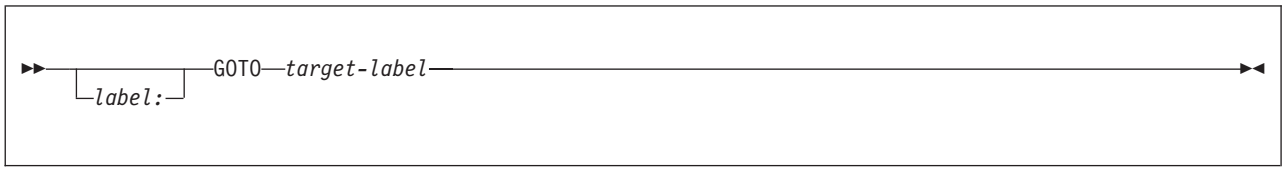
Effects of the statement: The GET DIAGNOSTICS statement does not change the contents of the diagnostics area except for DB2_GET_DIAGNOSTICS_DIAGNOSTICS.

Considerations for the SQLSTATE and SQLCODE SQL variables: The GET DIAGNOSTICS statement does not change the value of the SQLSTATE and SQLCODE SQL variables.

The stacked diagnostics area: The stacked diagnostics area is only available within a handler in a native SQL procedure and non-inline SQL functions.

GOTO statement

Syntax



Description

label
Specifies the label for the GOTO statement. The label name cannot be the same as the name of the SQL routine in which the label is used or another label in the same scope.

target-label
Specifies a label of the statement where processing is to continue. *target-label* must be defined as a label for an SQL procedure statement. The target label must be accessible to the GOTO statement as defined in "References to labels" on page 1965, subject to the following restrictions:

- If the GOTO statement is in a condition handler, *target-label* must be defined in that condition handler.
- If the GOTO statement is not defined in a condition handler, *target-label* must not be defined in a condition handler.

Notes

Using a GOTO statement: It is recommended that the GOTO statement be used sparingly. This statement interferes with the normal sequence of processing SQL statements, thus making a routine more difficult to read and maintain. Before using a GOTO statement, determine whether another statement, such as IF or LEAVE, can be used in place, to eliminate the need for a GOTO statement.

Effect on open cursors: When a GOTO statement transfers control out of a compound statement, all open cursors that are declared in the compound statement that contains the GOTO statement are closed, except cursors that are used to return result sets.

Examples

Example 1: In the following procedure, the GOTO statement branches outside of the current compound statement to a higher level:

```
CREATE PROCEDURE TESTGOTO5 ( )
P1: BEGIN
   DECLARE I ,A INTEGER;
     SET I = 1;
  LAB1: SET A = 1;
    BEGIN
      LAB2: SET A = 2;
         BEGIN
           SET I = I+1;
           IF I<3 THEN GOTO LAB1;
```

```
          END IF;
        END;
    END;
END P1
```

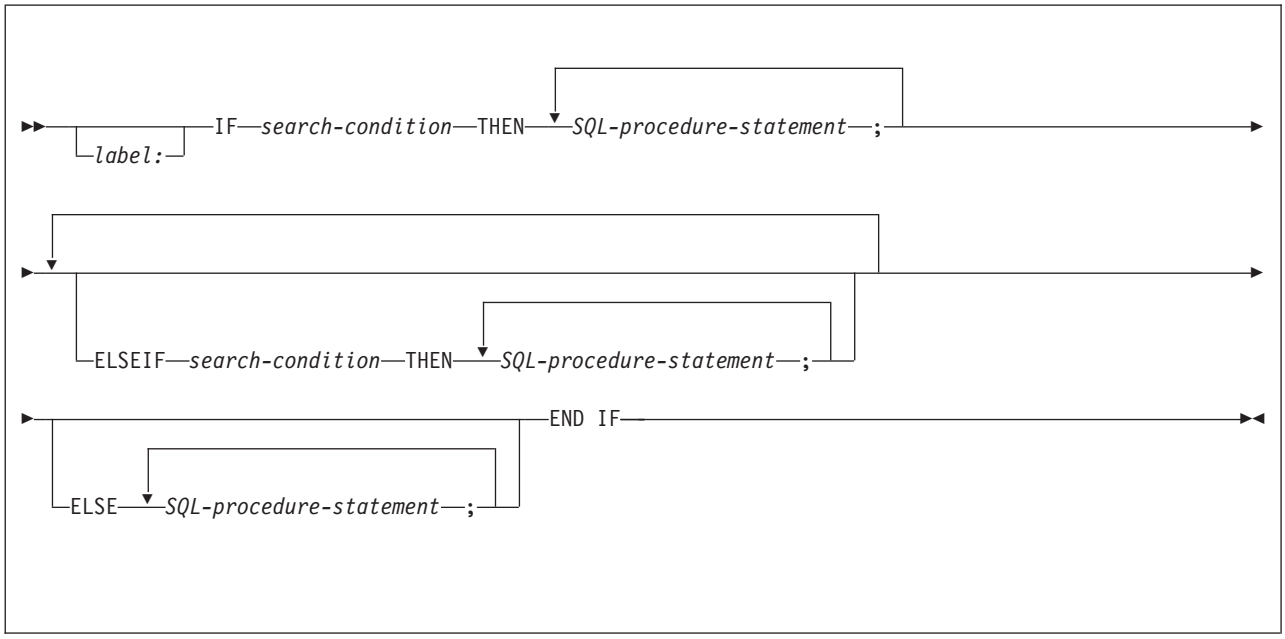
Example 2: In the following example, cursors are declared at multiple levels. The GOTO statement that specified TargLabel as the target label, results in the closing of cursors C1, C2, and C3. This is because cursors C1, C2, and C3 are all declared directly or indirectly in the compound statement with the label L1. The GOTO statement causes control to transfer out of the compound statement with label L1, so the cursors that are defined within that compound statement (at any level) are closed.

```
L0: BEGIN
      DECLARE CURSOR C0 ...
  ...
  TARGLABEL: ...
  ...
  L1: BEGIN
        DECLARE CURSOR C1 ...
        ...
      L2: BEGIN
            DECLARE CURSOR C2 ...
            ...
            GOTO TARGLABEL;
            ...
          L3: BEGIN
              DECALUE CURSOR C3 ...
            ...
          END L3;
      END L2;
   END L1;
END L0
```

IF statement

The IF statement executes different sets of SQL statements based on the result of search conditions.

Syntax



Description

label
Specifies the label for the IF statement. The label name cannot be the same as the name of the SQL routine or another label name within the same scope. For additional information, see "References to labels" on page 1965.

search-condition
Specifies the *search-condition* for which an SQL statement should be executed. If the condition is unknown or false, processing continues to the next search condition, until either a condition is true or processing reaches the ELSE clause.

SQL-procedure-statement
Specifies an SQL statement to be executed if the preceding *search-condition* is true. See "SQL-procedure-statement" on page 1968.

Examples

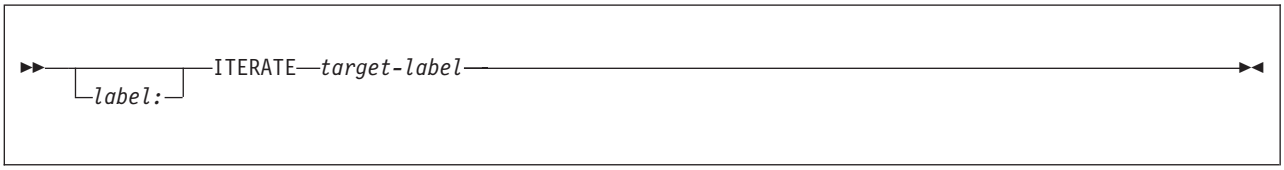
Assign a value to the SQL variable *new_salary* based on the value of SQL variable rating.

```
IF rating = 1
 THEN SET new_salary =
  new_salary + (new_salary * .10);
 ELSEIF rating = 2
  THEN SET new_salary =
   new_salary + (new_salary * .05);
 ELSE SET new_salary =
  new_salary + (new_salary * .02);
END IF
```

ITERATE statement

The ITERATE statement causes the flow of control to return to the beginning of a labeled loop.

Syntax



Description

label

Specifies the label for the ITERATE statement. The label name cannot be the same as the name of the SQL routine or another label within the same scope. For additional information, see "References to labels" on page 1965.

target-label

Specifies the label of the FOR, LOOP, REPEAT, or WHILE statement to which the flow of control is passed. *target-label* must be defined as a label for a FOR, LOOP, REPEAT, or WHILE statement. The ITERATE statement must be in that FOR, LOOP, REPEAT, or WHILE statement, or in the block of code that is directly or indirectly nested within that statement, subject to the following restrictions:

- If the ITERATE statement is in a condition handler, *target-label* must be defined in that condition handler.
- If the ITERATE statement is not in a condition handler, *target-label* must not be defined in a condition handler.

Examples

Example 1: This example uses a cursor to return information for a new department. If the not_found condition handler is invoked, the flow of control passes out of the loop. If the value of *v_dept* is 'D11', an ITERATE statement causes the flow of control to be passed back to the top of the LOOP statement. Otherwise, a new row is inserted into the table.

```
CREATE PROCEDURE ITERATOR ()
  LANGUAGE SQL
  MODIFIES SQL DATA
  BEGIN
    DECLARE v_dept CHAR(3);
    DECLARE v_deptname VARCHAR(29);
    DECLARE v_admdept CHAR(3);
    DECLARE at_end INTEGER DEFAULT 0;
    DECLARE not_found CONDITION FOR SQLSTATE '02000';
    DECLARE c1 CURSOR FOR
      SELECT deptno,deptname,admrdept
        FROM department
        ORDER BY deptno;
    DECLARE CONTINUE HANDLER FOR not_found
      SET at_end = 1;
    OPEN c1;
    ins_loop:
    LOOP
      FETCH c1 INTO v_dept, v_deptname, v_admdept;
      IF at_end = 1 THEN
```

```
      LEAVE ins_loop;
    ELSEIF v_dept = 'D11' THEN
      ITERATE ins_loop;
    END IF;
    INSERT INTO department (deptno,deptname,admrdept)
                    VALUES('NEW', v_deptname, v_admdept);
  END LOOP;
  CLOSE c1;
END
```

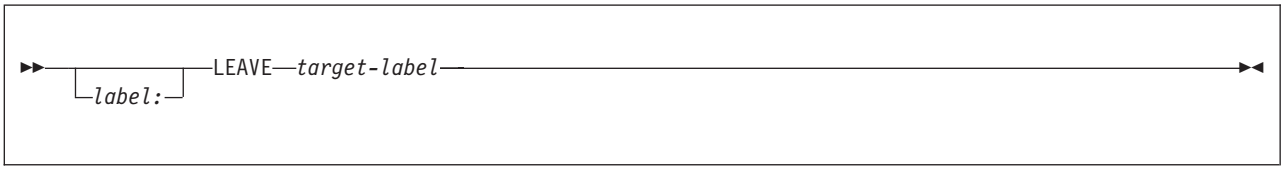
Example 2: An ITERATE statement can be issued from a nested block to cause that flow of control to return to the beginning of a loop at a higher level. In the following example, the ITERATE statement within the LAB2 compound statement causes the flow of control to return to the beginning of the LAB1 LOOP statement:

```
LAB1: LOOP
   SET A = 0;
   LAB2: BEGIN
     ...
     LAB3: BEGIN
        ...
        ITERATE LAB1;  -- Multilevel ITERATE
        ...
     END LAB3;
     ...
     ITERATE LAB1;   --  Multilevel ITERATE
     ...
   END LAB2;
END LOOP;S
```

LEAVE statement

The LEAVE statement transfers program control out of a FOR, LOOP, REPEAT, WHILE, or compound statement.

Syntax



Description

label
Specifies the label for the LEAVE statement. The label name cannot be the same as the name of the SQL routine or the same as another label that is within the same scope. For additional information, see "References to labels" on page 1965.

target-label
Specifies the label of the compound, FOR, LOOP, REPEAT, or WHILE statement to exit. *target-label* must be defined as a label for a compound, FOR, LOOP, REPEAT, or WHILE statement, or in a block of code that is directly or indirectly nested within that statement, subject to the following rules:

- If the LEAVE statement is in a condition handler, *target-label* must be defined in that condition handler.
- If the LEAVE statement is not in a condition handler, *target-label* must not be defined in a condition handler.

Notes

Effect on open cursors: When a LEAVE statement transfers control out of a compound statement, all open cursors in the compound statement, except cursors that are used to return result sets, are closed.

Examples

Example 1: The example contains a loop that fetches data for cursor c1. If the value of SQL variable *at_end* is not zero, the LEAVE statement transfers control out of the loop.

```
CREATE PROCEDURE LEAVE_LOOP (OUT COUNTER INTEGER)
  LANGUAGE SQL
  BEGIN
    DECLARE v_counter INTEGER;
    DECLARE v_firstnme VARCHAR(12);
    DECLARE v_midinit CHAR(1);
    DECLARE v_lastname VARCHAR(15);
    DECLARE at_end SMALLINT DEFAULT 0;
    DECLARE not_found CONDITION FOR SQLSTATE '02000';
    DECLARE c1 CURSOR FOR
      SELECT firstnme, midinit, lastname
      FROM employee;
    DECLARE CONTINUE HANDLER FOR not_found
      SET at_end = 1;
    SET v_counter = 0;
    OPEN c1;
```

```
   fetch_loop:
    LOOP
      FETCH c1 INTO v_firstnme, v_midinit, v_lastname;
      IF at_end <> 0 THEN
       LEAVE fetch_loop;
      END IF;
      SET v_counter = v_counter + 1;
    END LOOP fetch_loop;
    SET counter = v_counter;
    CLOSE c1;
  END
```

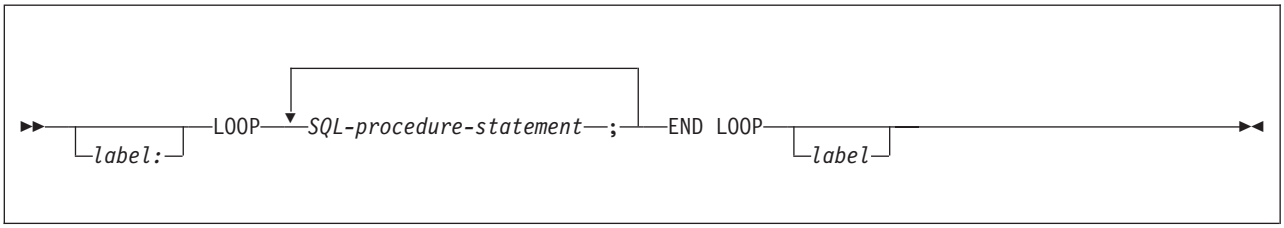
Example 2: A LEAVE statement can be issued from a nested block to leave a statement at a higher level. In the following example, the LEAVE statement within the LAB2 compound statement causes the LAB1 LOOP statement to terminate:

```
LAB1: LOOP
    ...
    LAB2: BEGIN
     SET A = 0;
     ...
     LAB3: BEGIN
        ...
        LEAVE LAB1;  -- Multilevel LEAVE
        ...
     END LAB3;
     ...
     LEAVE LAB1;   --  Multilevel LEAVE
     ...
   END LAB2;
END LOOP;S
```

LOOP statement

The LOOP statement executes a statement or group of statements multiple times.

Syntax



Description

label
Specifies the label for the LOOP statement. If the ending label is specified, a matching beginning label must be specified. A label name cannot be the same as the name of the SQL routine or another label within the same scope. For additional information, see "References to labels" on page 1965.

SQL-procedure-statement
Specifies an SQL statement to be executed in the loop. The statement must be one of the statements listed under "SQL-procedure-statement" on page 1968.

Notes

Considerations for the diagnostics area: At the beginning of the first iteration of the LOOP statement, and with every subsequent iteration, the diagnostics area is cleared.

Considerations for the SQLSTATE and SQLCODE SQL variables: Prior to executing the first SQL-procedure-statement within that LOOP statement, the SQLSTATE and SQLCODE values reflect the last values that were set prior to the LOOP statement. If the loop is terminated with a GOTO or a LEAVE statement, the SQLSTATE and SQLCODE values reflect successful completion of that statement. When the LOOP statement iterates, the SQLSTATE and SQLCODE values reflect the result of the last SQL statement that is executed within the LOOP statement.

Examples

This procedure uses a LOOP statement to fetch values from the employee table. Each time the loop iterates, the OUT parameter counter is incremented and the value of *v_midinit* is checked to ensure that the value is not a single space (' '). If *v_midinit* is a single space, the LEAVE statement passes the flow of control outside of the loop.

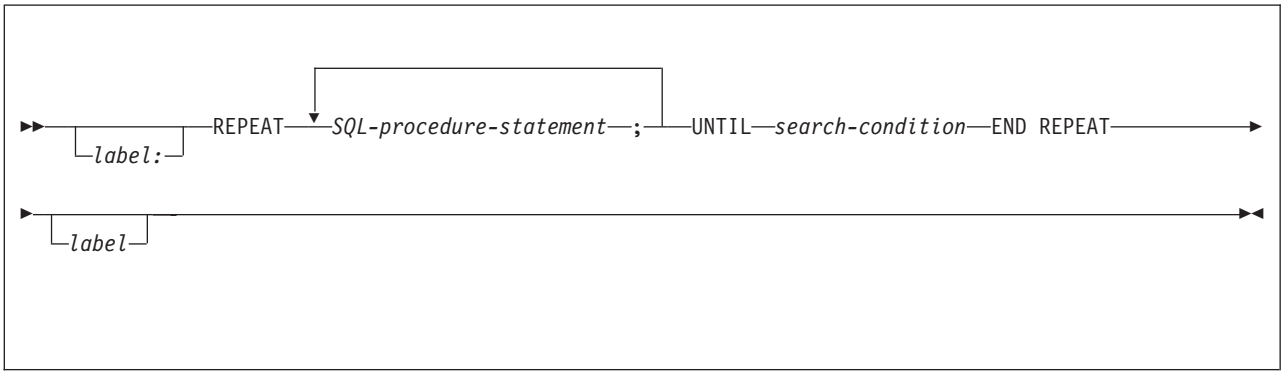
```
CREATE PROCEDURE LOOP_UNTIL_SPACE(OUT counter INTEGER)
   LANGUAGE SQL
   BEGIN
      DECLARE v_counter INTEGER DEFAULT 0;
      DECLARE v_firstnme VARCHAR(12);
      DECLARE v_midinit CHAR(1);
      DECLARE v_lastname VARCHAR(15);
      DECLARE c1 CURSOR FOR
         SELECT firstnme, midinit, lastname
           FROM employee;
      DECLARE EXIT HANDLER FOR NOT FOUND
```

```
      SET counter = -1;
   OPEN c1;
   fetch_loop:
   LOOP
     FETCH c1 INTO v_firstnme, v_midinit, v_lastname;
     IF v_midinit = ' ' THEN
        LEAVE fetch_loop;
     END IF;
     SET v_counter = v_counter + 1;
   END LOOP fetch_loop;
   SET counter = v_counter;
   CLOSE c1;
END
```

REPEAT statement

The REPEAT statement executes a statement or group of statements until a search condition is true.

Syntax



Description

label
Specifies the label for the REPEAT statement. If an ending label is specified, a matching beginning label must be specified. A label name cannot be the same as the name of the SQL routine or another label within the same scope. For additional information, see "References to labels" on page 1965.

SQL-procedure-statement
Specifies an SQL statement to be executed within the REPEAT loop. The statement must be one of the statements listed under "SQL-procedure-statement" on page 1968.

search-condition
Specifies a condition that is evaluated after each execution of the REPEAT loop. If the condition is true, the REPEAT loop will exit. If the condition is unknown or false, the looping continues.

Notes

Considerations for the diagnostics area: At the beginning of the first iteration of the REPEAT statement, and with every subsequent iteration, the diagnostics area is cleared.

Considerations for the SQLSTATE and SQLCODE SQL variables: At the beginning of the first iteration of the REPEAT statement, the SQLSTATE and SQLCODE values reflect the values that were set prior to the REPEAT statement. At the beginning of iterations 2 through *n* of the REPEAT statement, the SQLSTATE and SQLCODE SQL values reflect the result of evaluating the search condition in the UNTIL clause of that REPEAT. If the loop is terminated with a GOTO, ITERATE, or LEAVE statement, the SQLSTATE and SQLCODE values reflect the successful completion of that statement. Otherwise, after the END REPEAT of the REPEAT statement completes, the SQLSTATE and SQLCODE reflect the result of evaluating the search condition in the UNTIL clause of that REPEAT statement.

Examples

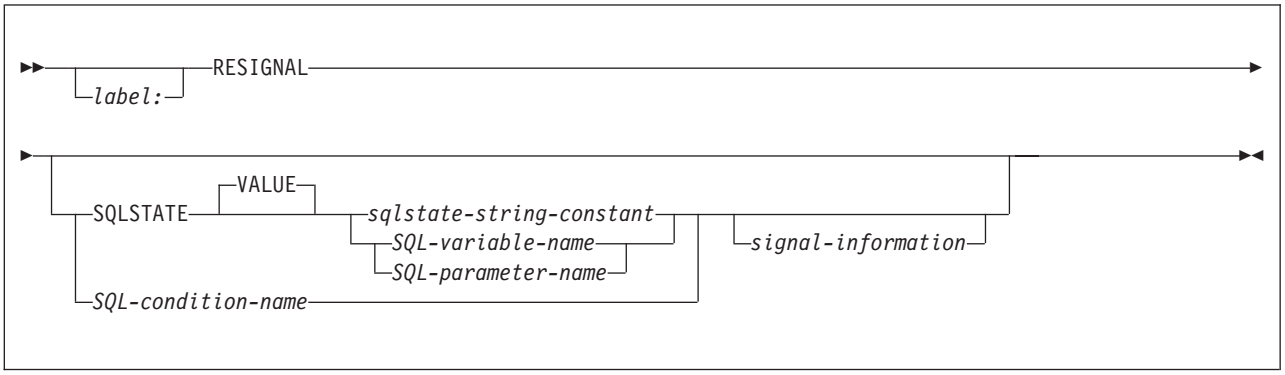
Use a REPEAT statement to fetch rows from a table.

```
fetch_loop:
REPEAT
 FETCH c1 INTO
  v_firstnme, v_midinit, v_lastname;
UNTIL
    SQLCODE <> 0
END REPEAT fetch_loop
```

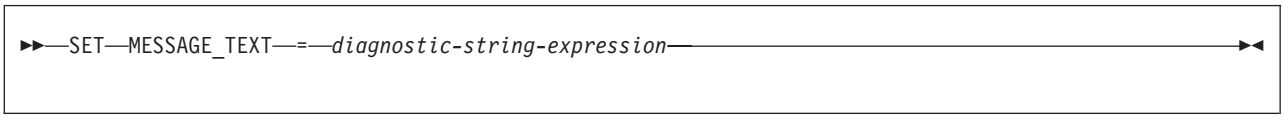
RESIGNAL statement

The RESIGNAL statement is used within a condition handler to resignal the condition that activated the handler, or to raise an alternate condition so that it can be processed at a higher level. It causes an exception, warning, or not found condition to be returned along with optional message text.

Syntax



signal-information:



Description

label
Specifies the label for the RESIGNAL statement. A label name cannot be the same as the name of the SQL routine or another label within the same scope. For additional information, see "References to labels" on page 1965.

SQLSTATE VALUE
Specifies the SQLSTATE that will be returned. Any valid SQLSTATE value can be used. It must be a character string constant with exactly five characters that follow the rules for SQLSTATE values:

- Each character must be from the set of digits ('0' through '9') or non-accented upper case letter ('A' through 'Z').
- The SQLSTATE class (the first two characters) cannot be '00' because it represents successful completion.

If the SQLSTATE does not conform to these rules, an error occurs.

sqlstate-string-constant
A character string constant with an actual length of five bytes that is a valid SQLSTATE value.

SQL-variable-name **or** *SQL-parameter-name*
Specifies an SQL variable or SQL parameter that is defined for the routine.

SQL-variable-name
Specifies an SQL variable that is declared within the *compound-statement* that contains the RESIGNAL statement or within a compound statement in which that compound statement is nested.

SQL-variable-name must be defined as CHAR or VARCHAR data type with an actual length of five bytes, must not be null, and must contain a valid SQLSTATE value.

SQL_parameter-name
Specifies an SQL parameter that is defined for the routine that contains the SQLSTATE value. The SQL parameter must be defined as CHAR or VARCHAR data type with an actual length of five bytes, must not be null, and must contain a valid SQLSTATE value.

SQL-condition-name
Specifies the name of the condition that will be returned. *SQL-condition-name* must be declared within the *compound-statement* that contains the RESIGNAL statement, or within a compound statement in which that compound statement is nested.

SET MESSAGE_TEXT
Specifies a string that describes the error or warning. The string is returned in the SQLERRMC field of the SQLCA or with the GET DIAGNOSTICS statement.

diagnostic-string-expression
An expression with a data type of CHAR or VARCHAR that returns a character string of up to 1000 bytes that describes the error or warning condition. For information on how to obtain the complete message text, see "GET DIAGNOSTICS" on page 1679.

Notes

While any valid SQLSTATE value can be used in the RESIGNAL statement, programmers should define new SQLSTATEs based on ranges reserved for applications. This practice prevents the unintentional use of an SQLSTATE value that might be defined by the database manager in a future release.

If the SQLSTATE or condition indicates that an exception is signaled (SQLSTATE class other than '01' or '02'):

- If a condition handler exists in the same compound statement as the RESIGNAL statement, and the compound statement contains a condition handler for SQLEXCEPTION or the specified SQLSTATE or condition, the exception is handled and control is transferred to that condition handler.
- If the compound statement is nested and an outer level compound statement has a condition handler for SQLEXCEPTION or the specified SQLSTATE or condition, the exception is handled and control is transferred to that condition handler.
- Otherwise, the exception is not handled and control is immediately returned to the end of the compound statement.

If an SQLSTATE or a condition indicates that a warning or a not found condition is signaled:

- If a condition handler exists in the same compound statement as the RESIGNAL statement, and the compound statement contains a condition handler for SQLWARNING, NOT FOUND, or the specified SQLSTATE or condition, the warning or not found condition is handled and control is transferred to that condition handler.
- If the compound statement is nested and an outer level compound statement contains a condition handler for SQLWARNING, NOT FOUND, or the specified

SQLSTATE or condition, the warning or not found condition is handled and control is returned to that condition handler.
- Otherwise, the warning is not handled and processing continues with the next statement.

Considerations for the diagnostics area: The RESIGNAL statement might modify the contents of the current diagnostics area. If an SQLSTATE or *condition-name* is specified as part of the RESIGNAL statement, the RESIGNAL statement starts with a clear diagnostics area and sets the RETURNED_SQLSTATE to reflect the specified SQLSTATE or *condition-name*. If message text is specified, the MESSAGE_TEXT item of the condition area is assigned the specified value. DB2_RETURNED_SQLCODE is set to +438 or -438 corresponding to the specified SQLSTATE or *condition-name*.

Processing a RESIGNAL statement: If the RESIGNAL statement is specified without an SQLSTATE clause or a *condition-name*, the SQL routine resignals the identical condition that invoked the handler and the SQLCODE is not changed.

When a RESIGNAL statement is issued and an SQLSTATE or *condition-name* is specified, the SQLCODE is based on he SQLSTATE value as follows:
- If the specified SQLSTATE class is either '01' or ‘02', a warning or not found is signaled and the SQLCODE is set to +438.
- Otherwise, an exception is returned and the SQLCODE is set to -438.

Examples

The following example detects a division by zero error. The IF statement uses a SIGNAL statement to invoke the overflow condition handler. The condition handler uses a RESIGNAL statement to return a different SQLSTATE to the client application.

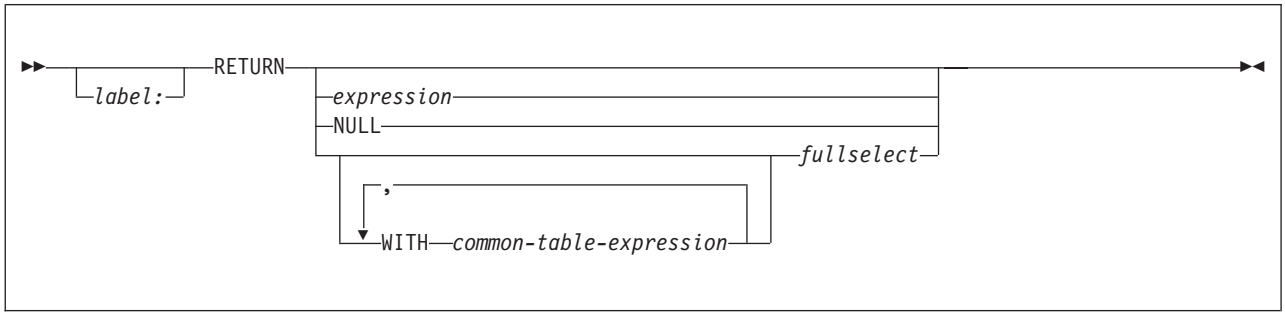
```
CREATE PROCEDURE divide (IN numerator INTEGER,
        IN denominator INTEGER,
        OUT divide_result INTEGER)
  LANGUAGE SQL
  CONTAINS SQL
  BEGIN
   DECLARE overflow CONDITION for SQLSTATE '22003';
   DECLARE CONTINUE HANDLER FOR overflow
     RESIGNAL SQLSTATE '22375';
   IF denominator = 0 THEN
     SIGNAL overflow;
   ELSE
     SET divide_result = numerator / denominator;
   END IF;
  END
```

RETURN statement

The RETURN statement is used to return from the routine.

- For an SQL scalar function, the scalar result of the function is returned. The body of an SQL scalar function must contain at least one RETURN statement and a RETURN statement must be executed when the function is invoked.
- For an SQL table function, the result table of the function is returned. A RETURN statement must be specified in the body of an SQL table function.
- For an SQL procedure, the RETURNS statement optionally returns an integer status value.

Syntax



Description

label

Specifies the label for the RETURN statement. A label name cannot be the same as the name of the SQL routine or another label within the same scope. For additional information, see "References to labels" on page 1965.

expression

Specifies a value that is returned from the routine.

- If the routine is a scalar function, the data type of the result must be assignable to the data type that is defined for the function result, using the storage assignment rules as described in "Assignment and comparison" on page 121. The RETURN statement must not contain a period specification.
- If the routine is a table function, a scalar expression (other than a scalar fullselect) cannot be specified. The data type of the result column of the fullselect must be assignable to the data type that is defined for the function result, using the storage assignment rules as described in "Assignment and comparison" on page 121. The RETURN statement must not contain a period specification.
- If the routine is a procedure, the data type of *expression* must be INTEGER.

NULL

The null value is returned from the SQL routine.

- If the routine is a scalar function, the null value is returned.
- If the routine is a table function, NULL must not be specified.
- If the routine is a procedure, NULL must not be specified.

WITH *common-table-expression*

Specifies one or more common table expressions that are to be used in the fullselect.

fullselect
Specifies the row or rows that are returned from the routine.

- If the routine is a scalar function, the fullselect must return one column and, at most, one row. The data type of the result column must be assignable to the data type that is defined for the function result, using the storage assignment rules as described in "Assignment and comparison" on page 121.
- If the routine is a table function, the fullselect can return zero or more rows with one or more columns. The number of columns in the fullselect must match the number of columns in the function result. In addition, the data types of the result table columns of the fullselect must be assignable to the data types of the columns that are defined for the function result, using the storage assignment rules as described in "Assignment and comparison" on page 121.
- If the routine is a procedure, fullselect must not be specified.

Notes

Considerations for SQL functions: A RETURN statement in an SQL function must specify *expression*, NULL, or *fullselect*. Only a single RETURN statement can be specified in the routine body of an SQL table function. The execution of an SQL function must end with a RETURN statement.

A data change table reference is not allowed in a RETURN statement in an SQL function.

Considerations for SQL procedures:

- **When a RETURN statement is used within an SQL procedure:** If a RETURN statement with a specified return value was used to return from a procedure, the SQLCODE, SQLSTATE, and message length in the SQLCA are initialized to zeros and the message text is set to blanks. An error is not returned to the caller.
- **When a RETURN statement is not used within an SQL procedure or when no value is specified:** If a RETURN statement was not used to return from a procedure or if a value is not specified on the RETURN statement, one of the following values is set:
  - If the procedure returns with an SQLCODE that is greater or equal to zero, the specified target for DB2_RETURN_STATUS in a GET DIAGNOSTICS statement will be set to a value of zero.
  - If the procedure returns with an SQLCODE that is less than zero, the specified target for DB2_RETURN_STATUS in a GET DIAGNOSTICS statement will be set to a value of '-1'.
- **When the value is returned from an SQL procedure:** When a value is returned from a procedure, the caller may access the value using one of the following methods:
  - The GET DIAGNOSTICS statement to retrieve the RETURN_STATUS when the SQL procedure was called from another SQL procedure.
  - The parameter bound for the return value parameter marker in the escape clause CALL syntax (?=CALL...) in a CLI application.
  - Directly from the SQLCA returned from processing the CALL of an SQL procedure by retrieving the value of sqlerrd[0]. When the SQLCODE is less than zero, the sqlerrd[0] value is not set. The application should assume a return status value of '-1'.

Examples

Example 1: Use a RETURN statement to return from an SQL procedure with a status value of zero if successful or '-200' if not successful.

```
BEGIN
       . . .
           GOTO FAIL;
       . . .
SUCCESS: RETURN 0;
   FAIL: RETURN -200;
END
```

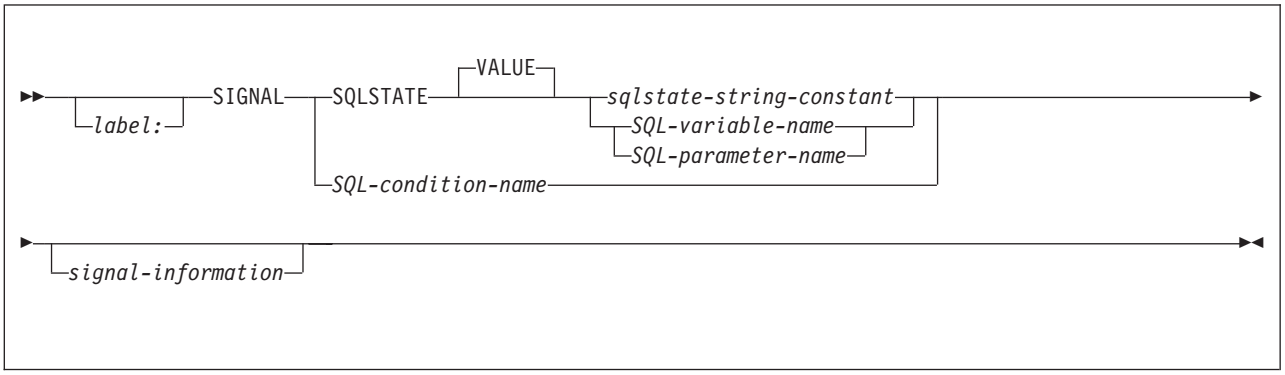
Example 2: Define a scalar function that returns the tangent of a value using the existing sine and cosine functions:

```
 CREATE FUNCTION TAN (X DOUBLE)
       RETURNS DOUBLE
       LANGUAGE SQL CONTAINS SQL NO EXTERNAL ACTION
       DETERMINISTIC
       RETURN SIN(x)/COS(x)
```

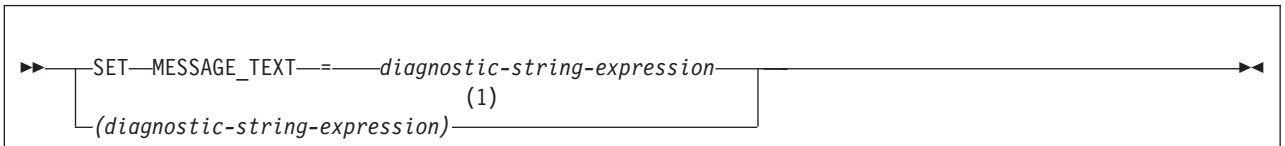
SIGNAL statement

The SIGNAL statement is used to return an exception or warning condition. It causes an error or warning to be returned with the specified SQLSTATE, along with optional message text. The SIGNAL statement places the specified condition information in the cleared diagnostics area.

Syntax



signal-information:



Notes:

1 *(diagnostic-string-expression)* must only be specified within a trigger body.

Description

label

Specifies the label for the SIGNAL statement. A label name cannot be the same as the name of the SQL routine or another label within the same scope. For additional information, see "References to labels" on page 1965.

SQLSTATE VALUE

Specifies the SQLSTATE that will be returned. Any valid SQLSTATE value can be used. It must be a character string constant with exactly five characters that follow the rules for SQLSTATEs:

- Each character must be from the set of digits ('0' through '9') or non-accented upper case letter ('A' through 'Z').
- The SQLSTATE class (the first two characters) cannot be '00' because it represents successful completion.

If the SQLSTATE does not conform to these rules, an error occurs.

sqlstate-string-constant

A character string constant with an actual length of five bytes that is a valid SQLSTATE value.

SQL-variable-name **or** *SQL-parameter-name*
Specifies an SQL variable or SQL parameter that contains a valid SQLSTATE value.

SQL-variable-name
Specifies an SQL variable that is declared within the *compound-statement* that contains the SIGNAL statement, or within a compound statement in which that compound statement is nested. *SQL-variable-name* must be defined as a CHAR or VARCHAR data type with an actual length of five bytes, must not be null, and must contain a valid SQLSTATE value.

SQL-parameter-name
Specifies an SQL parameter that is defined for the routine and contains the SQLSTATE value. The SQL parameter must be defined as a CHAR or VARCHAR data type with an actual length of five bytes, must not be null, and must contain a valid SQLSTATE value.

SQL-condition-name
Specifies the name of the condition that will be returned. The *SQL-condition-name* must be declared within the compound statement that contains the SIGNAL statement, or within a compound statement in which that compound statement is nested.

SET MESSAGE_TEXT
Specifies a string that describes the error or warning. The string is returned in the SQLERRMC field of the SQLCA or with the GET DIAGNOSTICS statement.

diagnostic-string-expression
An expression with a data type of CHAR or VARCHAR that returns a character string of up to 1000 bytes that describes the error or warning condition. For information on how to obtain the complete message text, see "GET DIAGNOSTICS" on page 1679.

(diagnostic-string-expression)
An expression with a data type of CHAR or VARCHAR that returns a character string of up to 1000 bytes that describes the error or warning condition. For information on how to obtain the complete message text, see "GET DIAGNOSTICS" on page 1679.

This syntax variation is only provided within the scope of a CREATE TRIGGER statement for compatibility with previous versions of DB2. To conform with the ANS and ISO standards, this form should not be used.

Notes

While any valid SQLSTATE value can be used in the SIGNAL statement, programmers should define new SQLSTATEs based on ranges reserved for applications. This practice prevents the unintentional use of an SQLSTATE value that might be defined by the database manager in a future release.

If the SQLSTATE or condition indicates that an exception is signaled:

- If a condition handler exists in the same compound statement as the SIGNAL statement, and the compound statement contains a condition handler for SQLEXCEPTION or the specified SQLSTATE or condition, the exception is handled and control is transferred to that condition handler.

- If the compound statement is nested and the outer level compound statement has a condition handler for SQLEXCEPTION or the specified SQLSTATE or condition, the exception is handled and control is transferred to that condition handler.
- Otherwise, the exception is not handled and control is immediately returned to the end of the compound statement.

If the SQLSTATE or condition indicates that a warning or not found condition is signaled:

- If a condition handler exists in the same compound statement as the SIGNAL statement, and the compound statement contains a condition handler for SQLWARNING, NOT FOUND, or the specified SQLSTATE or condition, the warning or not found condition is handled and control is transferred to that condition handler.
- If the compound statement is nested and an outer level compound statement contains a condition handler for SQLWARNING, NOT FOUND, or the specified SQLSTATE or condition, the warning or not found condition is handled and control is transferred to that condition handler.
- Otherwise, the warning or not found condition is not handled and processing continues with the next statement.

Considerations for the diagnostics area: The SIGNAL statement starts with a clear diagnostics area and sets the RETURNED_SQLSTATE to reflect the specified SQLSTATE or *condition-name*. If message text is specified, the MESSAGE_TEXT item of the condition area is assigned the specified value. DB2_RETURNED_SQLCODE is set to +438 or -438 corresponding to the specified SQLSTATE or *condition-name*.

Examples

Example 1: The following example shows an SQL procedure for an order system that signals an application error when a customer number is not known to the application. The ORDERS table includes a foreign key to the CUSTOMER table, requiring that the CUSTNO exist before an order can be inserted.

```
CREATE PROCEDURE SUBMIT_ORDER
            (IN ONUM INTEGER, IN CNUM INTEGER,
             IN PNUM INTEGER, IN QNUM INTEGER)
 LANGUAGE SQL
 SPECIFIC SUBMIT_ORDER
 MODIFIES SQL DATA
BEGIN
   DECLARE EXIT HANDLER FOR SQLSTATE VALUE '23503'
     SIGNAL SQLSTATE '75002'
         SET MESSAGE_TEXT = 'Customer number is not known';
   INSERT INTO ORDERS (ORDERNO, CUSTNO, PARTNO, QUANTITY)
      VALUES (ONUM, CNUM, PNUM, QNUM);
END
```

Example 2: The following example shows a trigger for an order system that allows orders to be recorded in an ORDERS table (ORDERNO, CUSTNO, PARTNO, QUANTITY) only if there is sufficient stock in the PARTS tables. When there is insufficient stock for an order, SQLSTATE '75001' is returned along with an appropriate error description.

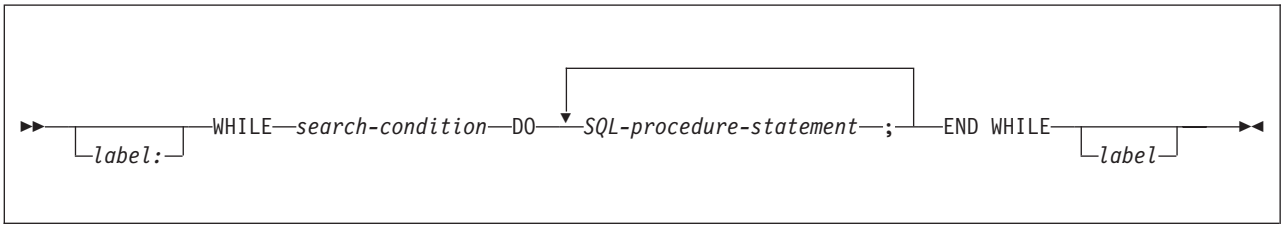
```
  CREATE TRIGGER CK_AVAIL
        NO CASCADE BEFORE INSERT ON ORDERS
        REFERENCING NEW AS NEW_ORDER
        FOR EACH ROW MODE DB2SQL
```

```
WHEN (NEW_ORDER.QUANTITY > (SELECT ON_HAND FROM PARTS
                            WHERE NEW_ORDER.PARTNO = PARTS.PARTNO))
  BEGIN ATOMIC
    SIGNAL SQLSTATE '75001' ('Insufficient stock for order');
  END
```

WHILE statement

The WHILE statement repeats the execution of a statement or group of statements while a specified condition is true.

Syntax



Description

label
Specifies the label for the WHILE statement. If the ending label is specified, it must be the same as the beginning label. A label name cannot be the same as the name of the SQL routine or another label within the same scope. For additional information, see "References to labels" on page 1965.

search-condition
Specifies a condition that is evaluated before each execution of the loop. If the condition is true, the SQL procedure statements in the loop are executed.

SQL-procedure-statement
Specifies a statement to be run within the WHILE loop. The statement must be one of the statements listed under "SQL-procedure-statement" on page 1968.

Notes

Considerations for the diagnostics area: At the beginning of the first iteration of the WHILE statement, and with every subsequent iteration, the diagnostics area is cleared.

Considerations for the SQLSTATE and SQLCODE SQL variables: With each iteration of the WHILE statement, when the first SQL-procedure-statement is executed, the SQLSTATE and SQLCODE SQL variables reflect the result of evaluating the search condition of that WHILE statement. If the loop is terminated with a GOTO, ITERATE, or LEAVE statement, the SQLSTATE and SQLCODE values reflect the successful completion of that statement. Otherwise, after the END WHILE of the WHILE statement completes, the SQLSTATE and SQLCODE reflect the result of evaluating that search condition of that WHILE statement.

Examples

Use a WHILE statement to fetch rows from a table while SQL variable at_end, which indicates whether the end of the table has been reached, is 0.

```
WHILE at_end = 0 DO
 FETCH c1 INTO
  v_firstnme, v_midinit,
  v_lastname, v_edlevel, v_salary;
 IF SQLCODE=100 THEN SET at_end=1;
 END IF;
END WHILE
```

Appendix. Additional information for DB2 SQL

These topics contain additional information for DB2 SQL.

Limits in DB2 for z/OS

DB2 for z/OS has system limits, object and SQL limits, length limits for identifiers and strings, and limits for certain data type values.

System storage limits might preclude the limits specified in this section. The limit for items not that are not specified below is limited by system storage.

The following table shows the length limits for identifiers.

Table 154. Identifier length limits. The term *byte(s)* in this table means the number of bytes for the UTF-8 representation unless noted otherwise.

Item	Limit
External-java-routine-name	1305 bytes
Name of an alias [1], auxiliary table, collection, clone table, constraint, correlation, cursor (except for DECLARE CURSOR WITH RETURN or the EXEC SQL utility), distinct type (both parts of two-part name), function (both parts of two-part name), host identifier, index, JARs, parameter, procedure, role, schema, sequence, specific, statement, storage group, savepoint, SQL condition, SQL label, SQL parameter, SQL variable, synonym, table, trigger, view, XML attribute name, XML element name	128 bytes
Name of an authorization ID or name of a security label.	8 bytes
Routine version identifier	64 EBCDIC bytes, and the UTF-8 representation of the name must not exceed 122 bytes.
Name of a column	30 bytes [1]
Name of cursor that is created with DECLARE CURSOR WITH RETURN	30 bytes
Name of cursor that is created with the EXEC SQL utility	8 bytes
Name of a location	16 bytes
Name of buffer pool name, catalog, database, plan, program, table space	8 bytes
Name of package	8 bytes (Only 8 EBCDIC characters are used for packages that are created with the BIND PACKAGE command. 128 bytes can be used for packages that are created as a result of the CREATE FUNCTION (SQL scalar) statement, the CREATE PROCEDURE (SQL - native) statement, the CREATE TRIGGER statement, or a BIND command that specifies a zFS file as DBRM library.)
Name of a profile that is created with CREATE TRUSTED CONTEXT or ALTER TRUSTED CONTEXT	127 bytes

Notes:

1. If the column name length or the distinct type schema or name length is greater than 30 Unicode bytes, truncation occurs in the sqlname field of the SQLDA when those objects are described in an application.

Table 155 shows the minimum and maximum limits for numeric values.

Table 155. Numeric limits

Item	Limit
Smallest SMALLINT value	-32768

Table 155. Numeric limits (continued)

Item	Limit
Largest SMALLINT value	32767
Smallest INTEGER value	-2147483648
Largest INTEGER value	2147483647
Smallest BIGINT value	-9223372036854775808
Largest BIGINT value	9223372036854775807
Smallest REAL value	About -7.2×10^{75}
Largest REAL value	About 7.2×10^{75}
Smallest positive REAL value	About 5.4×10^{-79}
Largest negative REAL value	About -5.4×10^{-79}
Smallest FLOAT value	About -7.2×10^{75}
Largest FLOAT value	About 7.2×10^{75}
Smallest positive FLOAT value	About 5.4×10^{-79}
Largest negative FLOAT value	About -5.4×10^{-79}
Smallest DECIMAL value	$1 - 10^{31}$
Largest DECIMAL value	$10^{31} - 1$
Largest decimal precision	31
Smallest DECFLOAT(16) value[1]	$-9.999999999999999 \times 10^{384}$
Largest DECFLOAT(16) value[1]	$9.999999999999999 \times 10^{384}$
Smallest positive DECFLOAT(16) value[1]	$1.000000000000000 \times 10^{-383}$
Largest negative DECFLOAT(16) value[1]	$-1.000000000000000 \times 10^{-383}$
Smallest DECFLOAT(34) value[1]	$-9.999999999999999999999999999999999 \times 10^{6144}$.
Largest DECFLOAT(34) value[1]	$9.999999999999999999999999999999999 \times 10^{6144}$.
Smallest positive DECFLOAT(34) value[1]	$1.000000000000000000000000000000000 \times 10^{-6143}$
Largest negative DECFLOAT(34) value[1]	$-1.000000000000000000000000000000000 \times 10^{-6143}$
Coefficient length for DECFLOAT values	DECFLOAT(16) is 16 digits; DECFLOAT(34) is 34 digits
Maximum Exponent (E_{max}) for DECFLOAT values	DECFLOAT(16) is 384; DECFLOAT(34) is 6144
Minimum Exponent (E_{min}) for DECFLOAT values	DECFLOAT(16) is -383; DECFLOAT(34) is -6143
Bias for DECFLOAT values	DECFLOAT(16) is 398; DECFLOAT(34) is 6176

Note:

1. These are the limits for normal numbers in DECFLOAT. DECFLOAT also contains special values such as NaN and Infinity that are also valid. DECFLOAT also supports subnormal numbers that are outside of the documented range.

The following table shows the length limits for strings.

Table 156. String length limits

Item	Limit
Maximum length of CHAR	255 bytes
Maximum length of GRAPHIC	127 double-byte characters
Maximum length of BINARY	255 bytes

Table 156. String length limits (continued)

Item	Limit
Maximum length[1] of VARCHAR	4046 bytes for 4 KB pages 8128 bytes for 8 KB pages 16320 bytes for 16 KB pages 32704 bytes for 32 KB pages
Maximum length of VARCHAR that can be indexed by an XML index	1000 bytes after conversion to UTF-8
Maximum length[1] of VARGRAPHIC	2023 double-byte characters for 4 KB pages 4064 double-byte characters for 8 KB pages 8160 double-byte characters for 16 KB pages 16352 double-byte characters for 32 KB pages
Maximum length of VARBINARY	32704 bytes
Maximum length of CLOB	2 147 483 647 bytes (2 GB - 1 byte)
Maximum length of DBCLOB	1 073 741 823 double-byte characters
Maximum length of BLOB	2 147 483 647 bytes (2 GB - 1 byte)
Maximum length of a character constant	32704 UTF-8 bytes
Maximum length of a hexadecimal character constant	32704 hexadecimal digits
Maximum length of a graphic string constant	16352 double-byte characters (32704 bytes when expressed in UTF-8)
Maximum length of a hexadecimal graphic string constant	32704 hexadecimal digits
Maximum length of a text string used for a scalar expression	4000 UTF-8 bytes
Maximum length of a concatenated character string	2 147 483 647 bytes (2 GB - 1 byte)
Maximum length of a concatenated graphic string	1 073 741 824 double-byte characters
Maximum length of a concatenated binary string	2 147 483 647 bytes (2 GB - 1 byte)
Maximum length of XML pattern text	4000 bytes after conversion to UTF-8
Maximum length of an XML element or attribute name in an XML document	1000 bytes
Maximum length of a namespace uri	1000 bytes
Maximum length of a namespace prefix	998 bytes
Largest depth of an internal XML tree	128 levels

Note:

1. The maximum length can be achieved only if the column is the only column in the table. Otherwise, the maximum length depends on the amount of space remaining on a page.

The following table shows the minimum and maximum limits for datetime values.

Table 157. Datetime limits

Item	Limit
Smallest DATE value (shown in ISO format)	0001-01-01
Largest DATE value (shown in ISO format)	9999-12-31
Smallest TIME value (shown in ISO format)	00.00.00
Largest TIME value (shown in ISO format)	24.00.00
Smallest TIMESTAMP WITHOUT TIME ZONE value	0001-01-01-00.00.00.000000000000

Table 157. Datetime limits (continued)

Item	Limit
Largest TIMESTAMP WITHOUT TIME ZONE value	9999-12-31-24.00.00.000000000000 [1]
Smallest TIMESTAMP WITH TIME ZONE value	0001-01-01-00.00.00.000000000000 +00:00
Largest TIMESTAMP WITH TIME ZONE value	9999-12-31-24.00.00.000000000000 +00:00 [1]
TIMESTAMP precision range	0 to 12
TIME ZONE hour range	-12 to 14
TIME ZONE minute range	0 to 59

Note:

1. The maximum value is stated as a UTC value. When a timestamp without a time zone is compared to a timestamp with time zone, a necessary adjustment is made using the implicit time zone. During that adjustment, the timestamp without time zone could be converted to a value that is greater than the maximum value for a timestamp with time zone value (this could occur on operations such as comparison and assignment). This situation can be avoided by using '9999-12-30-00.00.00.000000000000' as the maximum value for timestamp without time zone and '9999-12-30-00.00.00.000000000000 +00:00' as the maximum value for timestamp with time zone columns.

The following table shows the DB2 limits on SQL statements.

Table 158. DB2 limits on SQL statements

Item	Limit
Maximum number of columns that are in a table or view (the value depends on the complexity of the CREATE VIEW statement) or columns returned by a table function.	750 or fewer (including hidden columns) 749 if the table is a dependent
Maximum number of base tables in a view, SELECT, UPDATE, INSERT, MERGE, or DELETE	225
Maximum number of rows that can be inserted with a single INSERT or MERGE statement	32767
Maximum row and record sizes for a table	See the maximum record size table under CREATE TABLE.
Maximum number of volume IDs in a storage group	133
Maximum number of partitions in a partitioned table space or partitioned index	64 for table spaces that are not defined with LARGE or a DSSIZE greater than 2 GB. 4096, depending on what is specified for DSSIZE or LARGE and the page size.
Maximum sum of the lengths of limit key values of a partition boundary	765 UTF-8 bytes

Table 158. DB2 limits on SQL statements (continued)

Item	Limit
Maximum size of a partition (table space or index)	For table spaces that are not defined with LARGE or a DSSIZE greater than 2 GB: 4 GB, for 1 to 16 partitions 2 GB, for 17 to 32 partitions 1 GB, for 33 to 64 partitions For table spaces that are defined with LARGE or a DSSIZE of 4 GB: 4 GB, for 1 to 4096 partitions For table spaces that are defined with a DSSIZE greater than 4 GB: 256 GB, depending on the page size (for 1 to 64 partitions for 4 KB pages, for 1 to 128 partitions for 8 KB pages, for 1 to 256 partitions for 16 KB pages, and 1 to 512 partitions for 32 KB pages)
Maximum size of a non-partitioned index for a partitioned table space	For 5-byte EA table spaces: 16 TB for 4 KB pages 32 TB for 8 KB pages 64 TB for 16 KB pages 128 TB for 32 KB pages For table spaces that are defined with LARGE: 16 TB
Maximum length of an index key	Partitioning index: 255-*n* Nonpartitioning index that is padded: 2000-*n* Nonpartitioning index that is not padded: 2000-*n*-2*m* Where *n* is the number of columns in the key that allow nulls and *m* is the number of varying-length columns in the key
Maximum number of bytes used in the partitioning of a partitioned index	255 (This maximum limit is subject to additional limitations, depending on the number of partitions in the table space. The number of partitions * (106 + limit key size) must be less than 65394.)
Maximum number of columns in an index key	64
Maximum number of expressions in an index key	64
Maximum number of tables in a FROM clause	225 or fewer, depending on the complexity of the statement
Maximum number of subqueries in a statement	224
Maximum total length of host and indicator variables pointed to in an SQLDA	32767 bytes 2 147 483 647 bytes (2 GB - 1 byte) for a LOB, subject to the limitations that are imposed by the application environment and host language
Maximum size of application SQLDA for any statement that references host variables or parameter markers	99016 bytes
Maximum length of host variable used for insert or update operation	32704 bytes for a non-LOB 2 147 483 647 bytes (2 GB - 1 byte) for a LOB, subject to the limitations that are imposed by the application environment and host language
Maximum length of an SQL statement	2 097 152 bytes

Table 158. DB2 limits on SQL statements (continued)

Item	Limit
Maximum number of elements in a select list	750 or fewer, depending on whether the select list is for the result table of static scrollable cursor[1]
Maximum number of predicates in a WHERE or HAVING clause	Limited by storage
Maximum total length of columns of a query operation requiring a sort key (SELECT DISTINCT, ORDER BY, UNION, EXCEPT, and INTERSECT, without the ALL keyword, and the DISTINCT keyword for aggregate functions)	4032 bytes
Maximum total length of columns of a query operation requiring sort and evaluating column functions (MULTIPLE DISTINCT and GROUP BY)	65529 bytes
Maximum length of a sort key	16000 bytes
Maximum length of a check constraint	3800 bytes
Maximum number of bytes that can be passed in a single parameter of an SQL CALL statement	32765 bytes for a non-LOB 2 147 483 647 bytes (2 GB - 1 byte) for a LOB, subject to the limitations imposed by the application environment and host language
Maximum number of stored procedures, triggers, and user-defined functions that an SQL statement can implicitly or explicitly reference	64 nesting levels
Maximum length of the SQL path	2048 bytes
Maximum length of a WLM environment name in a CREATE PROCEDURE, CREATE FUNCTION, ALTER PROCEDURE, or ALTER FUNCTION statement.	32 bytes
Maximum number of XPath level in the XMLPATTERN clause of the CREATE INDEX statement.	50 nesting levels

Note:

1. If the scrollable cursor is read-only, the maximum number is 749 less the number of columns in the ORDER BY that are not in the select list. If the scrollable cursor is not read-only, the maximum number is 747.

The following table shows the DB2 system limits.

Table 159. DB2 system limits

Item	Limit
Maximum number of concurrent DB2 or application agents	Limited by the EDM pool size, buffer pool size, and the amount of storage that is used by each DB2 or application agent
Maximum size of a non-LOB table or table space	128 terabytes (TB)
Maximum size of a simple or segmented table space	64 GB
Maximum size of a log space	6-byte format: 2^{48} bytes 10-byte format: 2^{80} bytes
Maximum size of an active log data set	4 GB -1 byte
Maximum size of an archive log data set	4 GB -1 byte
Maximum number of active log copies	2
Maximum number of archive log copies	2

Table 159. DB2 system limits (continued)

Item	Limit
Maximum number of active log data sets (each copy)	93
Maximum number of archive log volumes (each copy)	10000
Maximum number of databases accessible to an application or user	Limited by system storage and EDM pool size
Maximum number of databases	65217
Maximum number of implicitly created databases	Maximum value of the sequence SYSIBM.DSNSEQ_IMPLICITDB, with a default of 10000
Maximum number of internal objects for each database [1]	32767
Maximum number of indexes on declared global temporary tables	10000
Maximum size of an EDM pool	The installation parameter maximum depends on available space
Maximum number of rows per page	255 for all table spaces except catalog and directory tables spaces, which have a maximum of 127
Maximum simple or segmented data set size	2 GB
Maximum partitioned data set size	See item "maximum size of a partition" in Table 158 on page 2015
Maximum LOB data set size	64 GB
Maximum number of table spaces that can be defined in a work file database	500
Maximum number of tables and triggers that can be defined in a work file database	11767

Note:

1. The number of internal object descriptors (OBDs) for external objects are as follows:
   - Table space: 2
   - Table: 1
   - Index: 2
   - Check constraint: 1
   - Referential integrity relationship: 2
   - Auxiliary relationship for each LOB column: 1
   - XML relationship for each XML column: 1
   - Trigger: 1
   - View that has an INSTEAD OF trigger: 1

Reserved schema names and reserved words

Restrictions exist on the use of certain names that are used by DB2. In some cases, names are reserved and cannot be used by application programs. In other cases, certain names are not recommended for use by application programs though not prevented by the database manager.

Reserved schema names

In general, for certain objects, schema names that begin with the prefix SYS are reserved. The schema name for these objects cannot begin with SYS except for certain exceptions.

The schema name for the objects listed in the following table must follow the restrictions listed in the table.

Recommendations:

- Do not to use SESSION name as a schema name.
- Do not use SYSPUBLIC as a schema name for a table or view.

Table 160. Objects with schema name restrictions and exceptions.

Object	Schema name restriction	Schema name exceptions
Distinct types	Cannot begin with SYS	The schema name can be: • SYSADM • SYSTOOLS[1]
User-defined functions	Cannot begin with SYS	The schema name can be: • SYSADM • SYSTOOLS[1] • SYSFUN[2]
Stored procedures	Cannot begin with SYS	The schema name can be: • SYSADM • SYSFUN[2] • SYSIBM • SYSIBMADM • SYSPROC • SYSTOOLS[1]
Sequences	Cannot begin with SYS	The schema name can be: • SYSADM
Triggers	Cannot begin with SYS	The schema name can be: • SYSADM • SYSTOOLS[1]
Column masks	Cannot begin with SYS	The schema name can be: • SYSADM
Row permissions	Cannot begin with SYS	The schema name can be: • SYSADM

Notes:

1. If the user who executes the CREATE statement has the SYSADM or SYSCTRL privilege.
2. For external user-defined scalar functions or external user-defined table functions if the user who executes the CREATE statement has the SYSADM or SYSCTRL privilege.

Reserved words

Certain words cannot be used as ordinary identifiers in some contexts because those words might be interpreted as SQL keywords. For example, ALL cannot be a column name in a SELECT statement. Each word, however, can be used as a delimited identifier in contexts where it otherwise cannot be used as an ordinary identifier. For example, if the quotation mark (") is the escape character that begins and ends delimited identifiers, "ALL" can appear as a column name in a SELECT statement.

New reserved words for this version of DB2 for z/OS are identified with notes in this topic. In addition, some topics in this information might indicate words that cannot be used in the specific context that is being described.

IBM SQL has additional reserved words that DB2 for z/OS does not enforce. Therefore, you should not use these additional reserved words as ordinary identifiers in names that have a continuing use. See *IBM DB2 SQL Reference for Cross-Platform Development* for a list of the words.

GUPI

ADD	AND	ASUTIME
AFTER	ANY	AT
ALL	AS	AUDIT
ALLOCATE	ARRAY[1]	AUX
ALLOW	ARRAY_EXISTS[1]	AUXILIARY
ALTER	ASENSITIVE	
	ASSOCIATE	

BEFORE
BEGIN
BETWEEN
BUFFERPOOL
BY

CALL	CLUSTER	CONTENT
CAPTURE	COLLECTION	CONTINUE
CASCADED	COLLID	CREATE
CASE	COLUMN	CURRENT
CAST	COMMENT	CURRENT_DATE
CCSID	COMMIT	CURRENT_LC_CTYPE
CHAR	CONCAT	CURRENT_PATH
CHARACTER	CONDITION	CURRENT_SCHEMA
CHECK	CONNECT	CURRENT_TIME
CLONE	CONNECTION	CURRENT_TIMESTAMP
CLOSE	CONSTRAINT	CURRVAL
	CONTAINS	CURSOR

DATA	DELETE	DO
DATABASE	DESCRIPTOR	DOCUMENT
DAY	DETERMINISTIC	DOUBLE
DAYS	DISABLE	DROP
DBINFO	DISALLOW	DSSIZE
DECLARE	DISTINCT	DYNAMIC
DEFAULT		

EDITPROC	ENDING	EXECUTE
ELSE	END-EXEC[2]	EXISTS
ELSEIF	ERASE	EXIT
ENCODING	ESCAPE	EXPLAIN
ENCRYPTION	EXCEPT	EXTERNAL
END	EXCEPTION	

FENCED	FOR
FETCH	FREE
FIELDPROC	FROM
FINAL	FULL
FIRST	FUNCTION

GENERATED	GRANT
GET	GROUP
GLOBAL	
GO	
GOTO	

HANDLER
HAVING
HOLD
HOUR
HOURS

IF	INHERIT	INSERT
IMMEDIATE	INNER	INTERSECT
IN	INOUT	INTO
INCLUSIVE	INSENSITIVE	IS
INDEX		ISOBID
		ITERATE

JAR
JOIN

KEEP
KEY

LABEL	LIKE	LOCK
LANGUAGE	LOCAL	LOCKMAX
LAST	LOCALE	LOCKSIZE
LC_CTYPE	LOCATOR	LONG
LEAVE	LOCATORS	LOOP
LEFT		

MAINTAINED	MINUTES
MATERIALIZED	MODIFIES
MICROSECOND	MONTH
MICROSECONDS	MONTHS
MINUTE	

NEXT	NOT	
NEXTVAL	NULL	
NO	NULLS	
NONE	NUMPARTS	
OBID	OPTIMIZE	
OF	OR	
OLD	ORDER	
ON	ORGANIZATION	
OPEN	OUT	
OPTIMIZATION	OUTER	
PACKAGE	PATH	PRIOR
PARAMETER	PIECESIZE	PRIQTY
PART	PERIOD	PRIVILEGES
PADDED	PLAN	PROCEDURE
PARTITION	PRECISION	PROGRAM
PARTITIONED	PREPARE	PSID
PARTITIONING	PREVVAL	PUBLIC
QUERY		
QUERYNO		
READS	RESULT_SET_LOCATOR	ROUND_DOWN
REFERENCES	RETURN	ROUND_FLOOR
REFRESH	RETURNS	ROUND_HALF_DOWN
RESIGNAL	REVOKE	ROUND_HALF_EVEN
RELEASE	RIGHT	ROUND_HALF_UP
RENAME	ROLE	ROUND_UP
REPEAT	ROLLBACK	ROW
RESTRICT	ROUND_CEILING	ROWSET
RESULT		RUN
SAVEPOINT	SET	STOGROUP
SCHEMA	SIGNAL	STORES
SCRATCHPAD	SIMPLE	STYLE
SECOND		SUMMARY
SECONDS	SOME	SYNONYM
SECQTY	SOURCE	SYSDATESYSTEM
SECURITY	SPECIFIC	SYSTIMESTAMP
SEQUENCE	STANDARD	
SELECT	STATIC	
SENSITIVE	STATEMENT	
SESSION_USER	STAY	
TABLE	TRUNCATE	
TABLESPACE	TYPE	
THEN		
TO		
TRIGGER		

UNDO	USER
UNION	USING
UNIQUE	
UNTIL	
UPDATE	

VALIDPROC	VCAT
VALUE	VERSIONING[1]
VALUES	VIEW
VARIABLE	VOLATILE
VARIANT	VOLUMES

WHEN
WHENEVER
WHERE
WHILE
WITH
WLM

XMLEXISTS
XMLNAMESPACES
XMLCAST

YEAR
YEARS

ZONE

Note:

1. New reserved word for Version 11.
2. COBOL only

GUPI

Characteristics of SQL statements in DB2 for z/OS

DB2 allows specific actions on each SQL statement, and only certain SQL statements are allowed in external routines and SQL procedures.

Actions allowed on SQL statements

Specific DB2 statements can be executed, prepared interactively or dynamically, or processed by the requester, the server, or the precompiler or coprocessor.

The following table shows whether a specific DB2 statement can be executed, prepared interactively or dynamically, or processed by the requester, the server, or the precompiler or coprocessor. The letter **Y** means *yes*.

Table 161. Actions allowed on SQL statements in DB2 for z/OS

SQL statement	Executable	Interactively or dynamically prepared	Processed by		
			Requesting system	Server	Precompiler or coprocessor
ALLOCATE CURSOR[1]	Y	Y	Y		
ALTER[2]	Y	Y		Y	
ASSOCIATE LOCATORS[1]	Y	Y	Y		
BEGIN DECLARE SECTION					Y
CALL[1]	Y			Y	
CLOSE	Y			Y	
COMMENT	Y	Y		Y	
COMMIT[8]	Y	Y		Y	
CONNECT	Y		Y		
CREATE[2]	Y	Y		Y	
DECLARE CURSOR					Y
DECLARE GLOBAL TEMPORARY TABLE	Y	Y		Y	
DECLARE STATEMENT					Y
DECLARE TABLE					Y
DECLARE VARIABLE					Y
DELETE	Y	Y		Y	
DESCRIBE prepared statement or table	Y			Y	
DESCRIBE CURSOR	Y		Y		
DESCRIBE INPUT	Y			Y	
DESCRIBE PROCEDURE	Y		Y		
DROP[2]	Y	Y		Y	
END DECLARE SECTION					Y
EXECUTE	Y			Y	
EXECUTE IMMEDIATE	Y			Y	
EXPLAIN	Y	Y		Y	
FETCH	Y			Y	
FREE LOCATOR[1]	Y	Y		Y	
GET DIAGNOSTICS	Y			Y	
GRANT[2]	Y	Y		Y	
HOLD LOCATOR[1]	Y	Y		Y	

Table 161. Actions allowed on SQL statements in DB2 for z/OS (continued)

SQL statement	Executable	Interactively or dynamically prepared	Processed by		
			Requesting system	Server	Precompiler or coprocessor
INCLUDE					Y
INSERT	Y	Y		Y	
LABEL	Y	Y		Y	
LOCK TABLE	Y	Y		Y	
MERGE	Y	Y		Y	
OPEN	Y			Y	
PREPARE	Y			Y[4]	
REFRESH TABLE	Y	Y		Y	
RELEASE connection	Y		Y		
RELEASE SAVEPOINT	Y	Y		Y	
RENAME[2]	Y	Y		Y	
REVOKE[2]	Y	Y		Y	
ROLLBACK[8]	Y	Y		Y	
SAVEPOINT	Y	Y		Y	
SELECT INTO	Y			Y	
SET CONNECTION	Y		Y		
SET CURRENT APPLICATION ENCODING SCHEME	Y		Y		
SET CURRENT DEBUG MODE	Y	Y		Y	
SET CURRENT DECFLOAT ROUNDING MODE	Y	Y		Y	
SET CURRENT DEGREE	Y	Y		Y	
SET CURRENT GET_ACCEL_ARCHIVE	Y	Y		Y	
SET CURRENT LC_CTYPE	Y	Y		Y	
SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION	Y	Y		Y	
SET CURRENT OPTIMIZATION HINT	Y	Y		Y	
SET CURRENT PACKAGE PATH	Y		Y		
SET CURRENT PACKAGESET	Y			Y	
SET CURRENT PRECISION	Y	Y		Y	
SET CURRENT QUERY ACCELERATION	Y	Y		Y	
SET CURRENT REFRESH AGE	Y	Y		Y	
SET CURRENT ROUTINE VERSION	Y	Y		Y	
SET CURRENT RULES	Y	Y		Y	
SET CURRENT SQLID[5]	Y	Y		Y	

Table 161. Actions allowed on SQL statements in DB2 for z/OS (continued)

SQL statement	Executable	Interactively or dynamically prepared	Processed by		
			Requesting system	Server	Precompiler or coprocessor
SET *host-variable* = CURRENT APPLICATION ENCODING SCHEME	**Y**	**Y**	**Y**		
SET *host-variable* = CURRENT DATE	**Y**			**Y**	
SET *host-variable* = CURRENT DEGREE	**Y**			**Y**	
SET *host-variable* = CURRENT MEMBER	**Y**			**Y**	
SET *host-variable* = CURRENT PACKAGESET	**Y**		**Y**		
SET *host-variable* = CURRENT PATH	**Y**			**Y**	
SET *host-variable* = CURRENT QUERY OPTIMIZATION LEVEL	**Y**			**Y**	
SET *host-variable* = CURRENT SERVER	**Y**		**Y**		
SET *host-variable* = CURRENT SQLID	**Y**			**Y**	
SET *host-variable* = CURRENT TIME	**Y**			**Y**	
SET *host-variable* = CURRENT TIMESTAMP	**Y**			**Y**	
SET *host-variable* = CURRENT TIMEZONE	**Y**			**Y**	
SET PATH	**Y**	**Y**		**Y**	
SET SCHEMA	**Y**	**Y**		**Y**	
SET *transition-variable* = CURRENT DATE	**Y**			**Y**	
SET *transition-variable* = CURRENT DEGREE	**Y**			**Y**	
SET *transition-variable* = CURRENT PATH	**Y**			**Y**	
SET *transition-variable* = CURRENT QUERY OPTIMIZATION LEVEL	**Y**			**Y**	
SET *transition-variable* = CURRENT SQLID	**Y**			**Y**	
SET *transition-variable* = CURRENT TIME	**Y**			**Y**	
SET *transition-variable* = CURRENT TIMESTAMP	**Y**			**Y**	
SET *transition-variable* = CURRENT TIMEZONE	**Y**			**Y**	
SIGNAL[6]	**Y**			**Y**	

Table 161. Actions allowed on SQL statements in DB2 for z/OS (continued)

SQL statement	Executable	Interactively or dynamically prepared	Processed by		
			Requesting system	Server	Precompiler or coprocessor
TRUNCATE	**Y**	**Y**		**Y**	
UPDATE	**Y**	**Y**		**Y**	
VALUES[6]	**Y**			**Y**	
VALUES INTO[7]	**Y**			**Y**	
WHENEVER					**Y**

Note:

1. The statement can be dynamically prepared. It cannot be issued dynamically.
2. The statement can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.
3. The statement can be dynamically prepared, but only from an ODBC or CLI driver that supports dynamic CALL statements.
4. The requesting system processes the PREPARE statement when the statement being prepared is ALLOCATE CURSOR or ASSOCIATE LOCATORS.
5. The value to which special register CURRENT SQLID is set is used as the SQL authorization ID for dynamic SQL statements only when DYNAMICRULES run behavior is in effect. The CURRENT SQLID value is ignored for the other DYNAMICRULES behaviors.
6. This statement can be used only in the triggered action of a trigger.
7. Local special registers can be referenced in a VALUES INTO statement if it results in the assignment of a single host-variable, not if it results in setting more than one value.
8. Some processing also occurs at the requester.

SQL statements allowed in external functions and stored procedures

Certain SQL statements can be executed in an external stored procedure or in an external user-defined function. Whether the statements can be executed depends on the level of SQL data access with which the stored procedure or external function is defined.

The following table shows which SQL statements in an external stored procedure or in an external user-defined function can execute. The letter **Y** means *yes*.

In general, if an executable SQL statement is encountered in a stored procedure or function defined as NO SQL, SQLSTATE 38001 is returned. If the routine is defined to allow some level of SQL access, SQL statements that are not supported in any context return SQLSTATE 38003. SQL statements not allowed for routines defined as CONTAINS SQL return SQLSTATE 38004, and SQL statements not allowed for READS SQL DATA return SQLSTATE 38002.

Table 162. SQL statements in external user-defined functions and stored procedures

	Level of SQL access			
SQL statement	**NO SQL**	**CONTAINS SQL**	**READS SQL DATA**	**MODIFIES SQL DATA**
ALLOCATE CURSOR			**Y**	**Y**
ALTER				**Y**
ASSOCIATE LOCATORS			**Y**	**Y**
BEGIN DECLARE SECTION	**Y**[1]	**Y**	**Y**	**Y**
CALL		**Y**[2]	**Y**[2]	**Y**[2]
CLOSE			**Y**	**Y**
COMMENT				**Y**
COMMIT[3]		**Y**	**Y**	**Y**
CONNECT		**Y**	**Y**	**Y**
CREATE				**Y**
DECLARE CURSOR	**Y**[1]	**Y**	**Y**	**Y**
DECLARE GLOBAL TEMPORARY TABLE				**Y**
DECLARE STATEMENT	**Y**[1]	**Y**	**Y**	**Y**
DECLARE TABLE	**Y**[1]	**Y**	**Y**	**Y**
DECLARE VARIABLE	**Y**[1]	**Y**	**Y**	**Y**
DELETE				**Y**
DESCRIBE			**Y**	**Y**
DESCRIBE CURSOR			**Y**	**Y**
DESCRIBE INPUT			**Y**	**Y**
DESCRIBE PROCEDURE			**Y**	**Y**
DROP				**Y**
END DECLARE SECTION	**Y**[1]	**Y**	**Y**	**Y**
EXECUTE		**Y**[4]	**Y**[4]	**Y**
EXECUTE IMMEDIATE		**Y**[4]	**Y**[4]	**Y**

Table 162. SQL statements in external user-defined functions and stored procedures (continued)

	Level of SQL access			
SQL statement	**NO SQL**	**CONTAINS SQL**	**READS SQL DATA**	**MODIFIES SQL DATA**
EXPLAIN				**Y**
FETCH			**Y**	**Y**
FREE LOCATOR		**Y**	**Y**	**Y**
GET DIAGNOSTICS		**Y**	**Y**	**Y**
GRANT				**Y**
HOLD LOCATOR		**Y**	**Y**	**Y**
INCLUDE	**Y**[1]	**Y**	**Y**	**Y**
INSERT				**Y**
LABEL				**Y**
LOCK TABLE		**Y**	**Y**	**Y**
MERGE				**Y**
OPEN			**Y**	**Y**
PREPARE		**Y**	**Y**	**Y**
REFRESH TABLE				**Y**
RELEASE connection		**Y**	**Y**	**Y**
RELEASE SAVEPOINT[6]				**Y**
REVOKE				**Y**
ROLLBACK[6, 7, 8]		**Y**	**Y**	**Y**
ROLLBACK TO SAVEPOINT[6, 7, 8]				**Y**
SAVEPOINT[6]				**Y**
SELECT INTO			**Y**	**Y**
SET CONNECTION		**Y**	**Y**	**Y**
SET CURRENT DEBUG MODE		**Y**	**Y**	**Y**
SET CURRENT ROUTINE VERSION		**Y**	**Y**	**Y**
SET host-variable Assignment		**Y**[5]	**Y**	**Y**
SET special register		**Y**	**Y**	**Y**
SET transition-variable Assignment		**Y**[5]	**Y**	**Y**
SIGNAL		**Y**	**Y**	**Y**
TRUNCATE				**Y**
UPDATE				**Y**
VALUES			**Y**	**Y**
VALUES INTO		**Y**[5]	**Y**	**Y**
WHENEVER	**Y**[1]	**Y**	**Y**	**Y**

Table 162. SQL statements in external user-defined functions and stored procedures (continued)

	Level of SQL access			
SQL statement	**NO SQL**	**CONTAINS SQL**	**READS SQL DATA**	**MODIFIES SQL DATA**

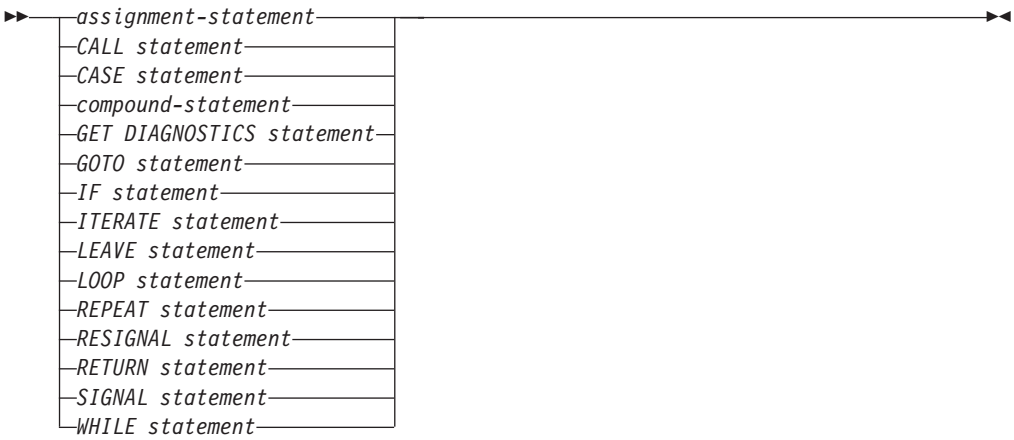
Notes:

1. Although the SQL option implies that no SQL statements can be specified, non-executable statements are not restricted.
2. The stored procedure that is called must have the same or more restrictive level of SQL data access than the current level in effect. For example, a routine defined as MODIFIES SQL DATA can call a stored procedure defined as MODIFIES SQL DATA, READS SQL DATA, CONTAINS SQL, or NO SQL. A routine defined as CONTAINS SQL can call a procedure defined as CONTAINS SQL or NO SQL.
3. The COMMIT statement cannot be executed in a user-defined function. The COMMIT statement cannot be executed in a stored procedure if the procedure is in the calling chain of a user-defined function or trigger.
4. The statement specified for the EXECUTE statement must be a statement that is allowed for the particular level of SQL data access in effect. For example, if the level in effect is READS SQL DATA, the statement must not be an INSERT, UPDATE, MERGE, or DELETE statement.
5. The statement is supported only if it does not contain a subquery or query-expression.
6. RELEASE SAVEPOINT, SAVEPOINT, and ROLLBACK (with the TO SAVEPOINT clause) cannot be executed from a user-defined function.
7. If the ROLLBACK statement (without the TO SAVEPOINT clause) is executed in a user-defined function, an error is returned to the calling program, and the application is placed in a *must rollback* state.
8. The ROLLBACK statement (without the TO SAVEPOINT clause) cannot be executed in a stored procedure if the procedure is in the calling chain of a user-defined function or trigger.

SQL control statements for external SQL procedures

SQL control statements for external SQL procedures can be used only with SQL procedures that are created with the FENCED or EXTERNAL clause. *SQL control statements* provide the capability to control the logic flow, declare and set variables, and handle warnings and exceptions. Some SQL control statements include other nested SQL statements.

SQL-control-statement:



Control statements are supported in SQL procedures. External SQL procedures are created by specifying either FENCED or EXTERNAL, LANGUAGE SQL, and an SQL routine body on the "CREATE PROCEDURE (SQL - external)" on page 1338 statement. The SQL routine body must be a single SQL statement which may be an SQL control statement.

The remainder of this chapter contains a description of the control statements that are supported for external SQL procedures, and includes syntax diagrams, semantic descriptions, usage notes, and examples of the use of the statements that constitute the SQL routine body. In addition, you can find information about referencing SQL parameters and variables in "References to SQL parameters and SQL variables."

The two common elements that are used in describing specific SQL control statements are:

- SQL control statements as described above
- "SQL-procedure-statement" on page 2035

References to SQL parameters and SQL variables

SQL parameters and SQL variables can be referenced anywhere in the statement where an expression or a host variable can be specified. Host variables can be specified in SQL routines. SQL parameters and SQL variables can be referenced anywhere in the compound statement in which they are declared and can be qualified with the label name that is specified at the beginning of the compound statement.

All SQL parameters and SQL variables are considered nullable. The name of an SQL parameter or SQL variable in an SQL routine can be the same as the name of a column in a table or view that the SQL routine references. Names that are the same should be explicitly qualified. Qualifying a name clearly indicates whether the name refers to a column, SQL variable, or SQL parameter.

If the name is not qualified, the following rules describe whether the name refers to the column, the SQL variable, or the SQL parameter:

- The name is checked first as an SQL variable name and then as an SQL parameter name.
- If an SQL variable or SQL parameter by that name is not found, the name is assumed to be a column name.

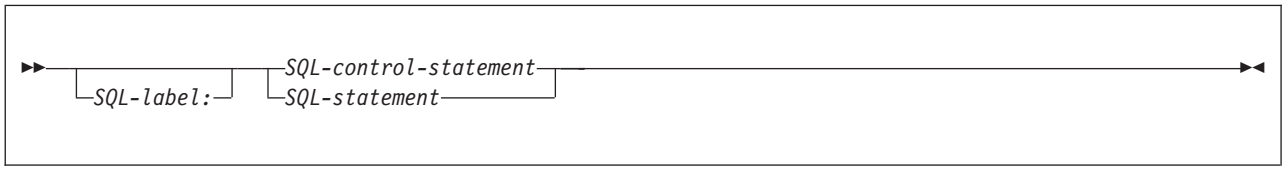
The name of an SQL variable or SQL parameter in an SQL routine can be the name of an identifier that is used in certain SQL statements. If the name is not qualified, the following rules describe whether the name refers to the identifier, the SQL variable, or the SQL parameter:

- In the SET PATH and SET SCHEMA statements, the name is checked as an SQL variable name or an SQL parameter name. If an SQL variable or SQL parameter by that name is not found, the name is assumed to be an identifier.
- In the ASSOCIATE LOCATORS, CONNECT statement, the SET CONNECTION statement, and the RELEASE (connection) statement the name is used as an identifier.

SQL-procedure-statement

An SQL control statement may allow multiple SQL statements to be specified within the SQL control statement. These statements are defined as SQL procedure statements.

Syntax



Description

SQL-label
Specifies a label for the statement. *SQL-label* must not be a delimited identifier that includes lowercase letters or special characters. The label must be unique within the procedure.

SQL-control-statement
Specifies an SQL statement that provides the capability to control logic flow, declare and set variables, and handle warnings and exceptions, as defined in this section. Control statements are supported in SQL procedures.

SQL-statement
Specifies an SQL statement. These statements are described in Chapter 5, "Statements," on page 833.

Notes

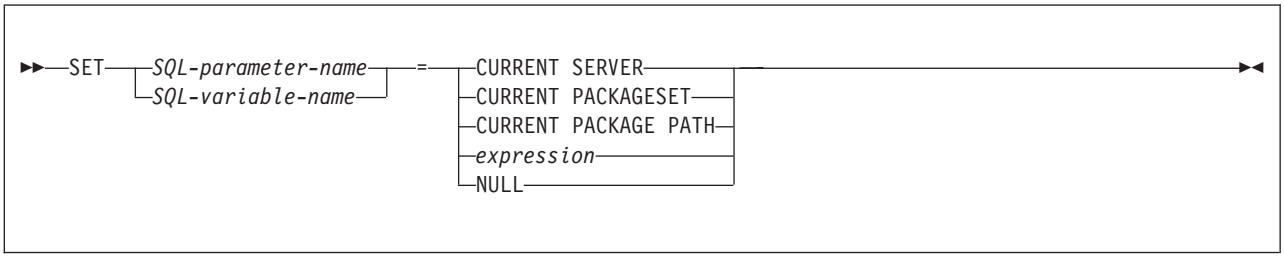
Comments: Comments can be included within the body of an SQL procedure. In addition to the double-dash form of comments (--), a comment can begin with /* and end with */. The following rules apply to this form of comment:
- The beginning characters /* must be adjacent and on the same line.
- The ending characters */ must be adjacent and on the same line.
- Comments can be started wherever a space is valid.
- Comments can be continued to the next line.

Handling errors and warnings: Conditions can be detected within an SQL procedure by using the following methods:
- Test the special SQL variables SQLSTATE and SQLCODE.
- Issue a GET DIAGNOSTICS statement to request the condition information. See "GET DIAGNOSTICS" on page 1679.
- Define condition handlers to detect and process conditions. See "compound-statement" on page 2043 for information about defining condition handlers.

assignment-statement (SQL control statements for external routines)

The assignment statement assigns a value to an SQL parameter or to an SQL variable.

Syntax



Description

SQL-parameter-name
Identifies the parameter that is the assignment target. The parameter must be specified in *parameter-declaration* in the CREATE PROCEDURE statement and must be defined as OUT or INOUT.

SQL-variable-name
Identifies the SQL variable that is the assignment target. SQL variables can be declared in a compound-statement and must be declared before it is used. For information on declaring SQL variables, see "compound-statement" on page 2043.

expression **or NULL**
Specifies the expression or value that is the assignment source. The expression can be any expression of the type described in "Expressions" on page 240 except it cannot contain a reference to local special registers (CURRENT SERVER, CURRENT PACKAGESET, or CURRENT PACKAGE PATH).

Notes

Assignment rules: Assignment statements in SQL procedures must conform to the SQL assignment rules. For example, the data type of the target and source must be compatible. See "Assignment and comparison" on page 121 for assignment rules.

When a string is assigned to a fixed-length variable and the length of the string is less than the length attribute of the target, the string is padded on the right with the necessary number of single-byte or double-byte blanks. When a string is assigned to a variable and the string is longer than the length attribute of the variable, the value is truncated and a warning is returned.

The ENCODING bind option is not used during processing of assignments to string variables. For example, assume that the system does not use mixed or DBCS, and the system EBCDIC SBCS CCSID is 37. Character conversion will not occur on assignment even if CCSID 500 is specified for the ENCODING bind parameter for the package for the procedure.

If truncation of the whole part of a number occurs on assignment to a numeric variable, the value is truncated and a warning is returned.

Assignments involving SQL parameters:

- An IN parameter can appear on the left side of an assignment statement. When control returns to the caller, the original value of an IN parameter is passed to the caller.
- An OUT parameter can appear on the left or right side of an assignment statement. When control returns to the caller, the last value that is assigned to an OUT parameter is returned to the caller.
- An INOUT parameter can appear on the left or right side of an assignment statement. The first value of the parameter is determined by the caller, and the last value that is assigned to the parameter is returned to the caller.
- A LOB parameter can not be used as an output value in an SQL statement in an SQL procedure when connected to a remote site. To circumvent the restriction, use a LOB SQL variable instead of a LOB parameter.

Considerations for SQLSTATE and SQLCODE SQL variables: Assignment to these variables is not prohibited. However, it is not recommended as assignment does not affect the diagnostic area or result in the activation of condition handlers. Furthermore, processing an assignment to these SQL variables causes the specified values for the assignment to be overlayed with the SQL return codes returned from executing the statement that does the assignment.

Examples

Increase the SQL variable *p_salary* by 10 percent.

```
SET p_salary = p_salary + (p_salary * .10)
```

Set SQL variable *p_salary* to the null value.

```
SET p_salary = NULL
```

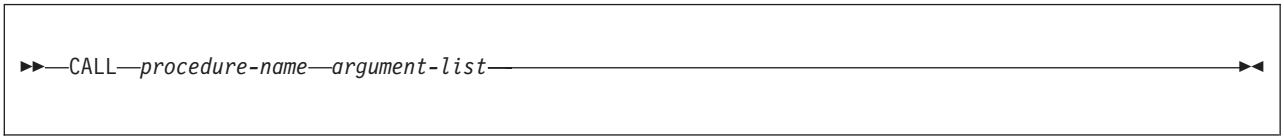
Set SQL variable *midinit* to the first character of SQL variable *midname*.

```
SET midinit = SUBSTR(midname,1,1)
```

CALL statement

The CALL statement invokes a stored procedure.

Syntax



argument-list:

Description

procedure-name

Identifies the stored procedure to call. The procedure name must identify a stored procedure that exists at the current server.

argument-list

Identifies a list of values to be passed as parameters to the stored procedure. The number of parameters must be the same as the number of parameters defined for the stored procedure. See "CALL" on page 1117 for more information.

Control is passed to the stored procedure according to the calling conventions for SQL procedures. When execution of the stored procedure is complete, the value of each parameter of the stored procedure is assigned to the corresponding parameter of the CALL statement defined as OUT or INOUT.

SQL-variable-name

Specifies an SQL variable as an argument to the stored procedure. For an explanation of references to SQL variables, see "References to SQL parameters and SQL variables" on page 2033.

SQL-parameter-name

Specifies an SQL parameter as an argument to the stored procedure. For an explanation of references to SQL parameters, see "References to SQL parameters and SQL variables" on page 2033.

expression

The parameter is the result of the specified *expression*, which is evaluated before the stored procedure is invoked. If *expression* is a single *SQL-parameter-name* or *SQL-variable-name*, the corresponding parameter of the procedure can be defined as IN, INOUT, or OUT. Otherwise, the corresponding parameter of the procedure must be defined as IN. If the result of the *expression* can be the null value, either the description of the

procedure must allow for null parameters or the corresponding parameter of the stored procedure must be defined as OUT.

The following additional rules apply depending on how the corresponding parameter was defined in the CREATE PROCEDURE statement for the procedure:

- IN *expression* can contain references to multiple SQL parameters or variables. In addition to the rules stated in "Expressions" on page 240 for *expression*, *expression* cannot include a column name, an aggregate function, or a user-defined function that is sourced on an aggregate function.
- INOUT or OUT *expression* can only be a single SQL parameter or variable.

NULL
The parameter is a null value. The corresponding parameter of the procedure must be defined as IN and the description of the procedure must allow for null parameters.

Notes

See "CALL" on page 1117 for more information on the SQL CALL statement.

Examples

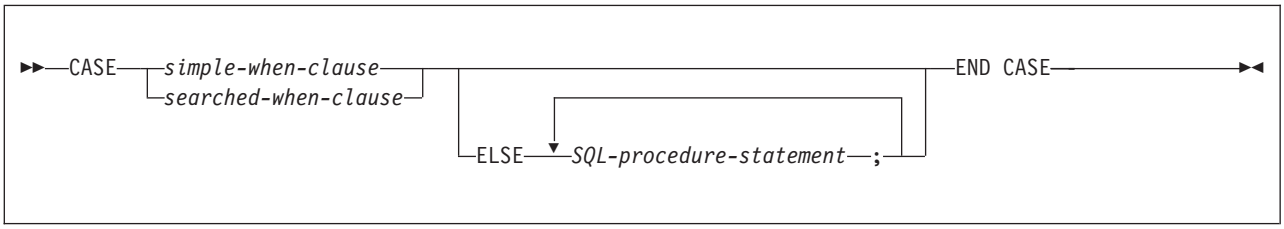
Call stored procedure proc1 and pass SQL variables as parameters.

```
CALL proc1(v_empno, v_salary)
```

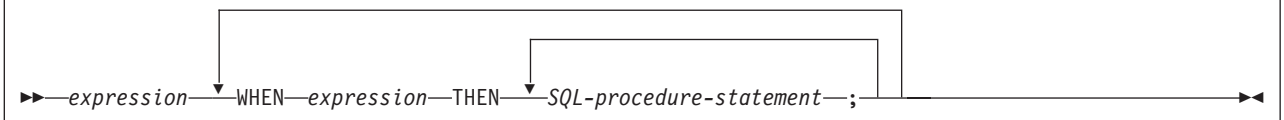
CASE statement

The CASE statement selects an execution path based on the evaluation of one or more conditions. A CASE statement operates in the same way as a CASE expression.

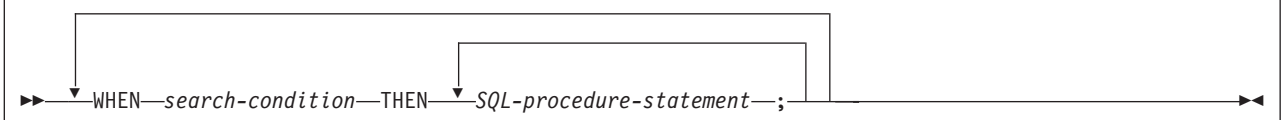
Syntax



simple-when-clause:



searched-when-clause:



Description

CASE
Begins a *case-expression*.

simple-when-clause
Specifies the *expression* prior to the first WHEN keyword that is tested for equality with the value of each *expression* that follows the WHEN keyword, and the result to be executed when those expressions are equal. If the comparison is true, the THEN statement is executed. If the result is unknown or false, processing continues to the next expression or the ELSE statement.

The data type of the *expression* prior to the first WHEN keyword must be comparable to the data types of each *expression* that follows the WHEN keywords.

searched-when-clause
Specifies the *search-condition* that is applied to each row or group of table data presented for evaluation, and the result when that condition is true. *search-condition* cannot contain a *fullselect*. If the search condition is true, the

THEN statement is executed. If the condition is unknown or false, processing continues to the next search condition or the ELSE statement.

SQL-procedure-statement
Specifies a statement that follows the THEN and ELSE keyword. The statement specifies the result of a *searched-when-clause* or a *simple-when-clause* that is true, or the result if no case is true. The statement must be one of the statements listed under "SQL-procedure-statement" on page 2035.

search-condition
Specifies a condition that is true, false, or unknown about a row or group of table data.

ELSE *SQL-procedure-statement*
If none of the conditions specified in the *simple-when-clause* or *searched-when-clause* are true, the statements in the *else-clause* are executed.

If none of the conditions specified in the WHEN clause are true and an ELSE clause is not specified, an error is returned at run time, and the execution of the CASE statement is terminated.

END CASE
Ends a *case-statement*.

Notes

If none of the conditions specified in the WHEN clause are true and an ELSE clause is not specified, an error is returned at run time, and the execution of the CASE statement is terminated.

CASE statements that use a simple case statement WHEN clause can be nested up to three levels. CASE statements that use a searched statement WHEN clause have no limit to the number of nesting levels.

Considerations for the SQLSTATE and SQLCODE SQL variables: When the first SQL-procedure-statement in the CASE statement is executed, the SQLSTATE and SQLCODE SQL variables reflect the result of evaluating the expression or search conditions of that CASE statement. If a CASE statement does not include an ELSE clause and none of the search conditions evaluate to true, an error is returned.

Examples

Example 1: Use a simple case statement WHEN clause to update column DEPTNAME in table DEPT, depending on the value of SQL variable *v_workdept*.

```
CASE v_workdept
 WHEN 'A00'
  THEN UPDATE DEPT SET
   DEPTNAME = 'DATA ACCESS 1';
 WHEN 'B01'
  THEN UPDATE DEPT SET
   DEPTNAME = 'DATA ACCESS 2';
 ELSE UPDATE DEPT SET
   DEPTNAME = 'DATA ACCESS 3';
END CASE
```

Example 2: Use a searched case statement WHEN clause to update column DEPTNAME in table DEPT, depending on the value of SQL variable *v_workdept*.

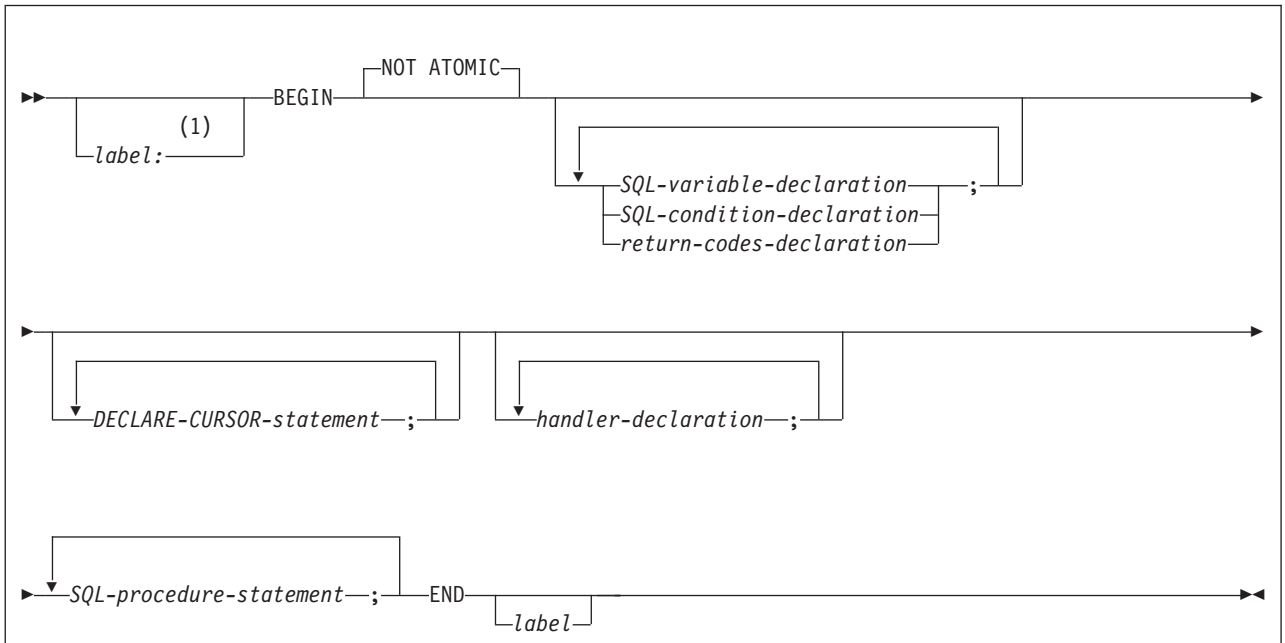
```
CASE
 WHEN v_workdept < 'B01'
  THEN UPDATE DEPT SET
```

```
    DEPTNAME = 'DATA ACCESS 1';
  WHEN v_workdept < 'C01'
   THEN UPDATE DEPT SET
    DEPTNAME = 'DATA ACCESS 2';
  ELSE UPDATE DEPT SET
    DEPTNAME = 'DATA ACCESS 3';
END CASE
```

compound-statement

A compound statement contains a group of statements and declarations for SQL variables, cursors, and condition handlers.

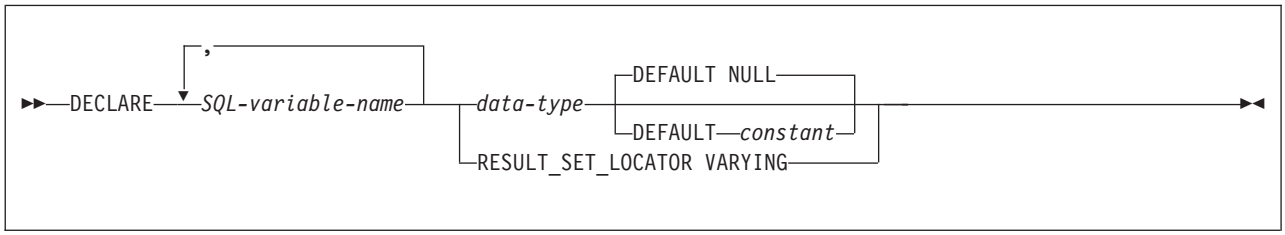
Syntax



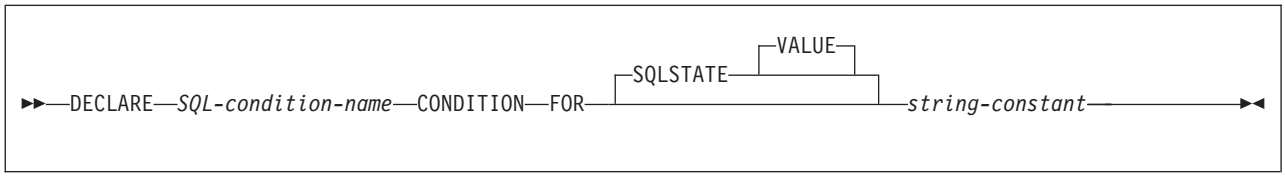
Notes:

1 Only one *label:* can be specified for each *SQL-procedure-statement*. If an ending label is specified for this beginning label, the labels must be the same.

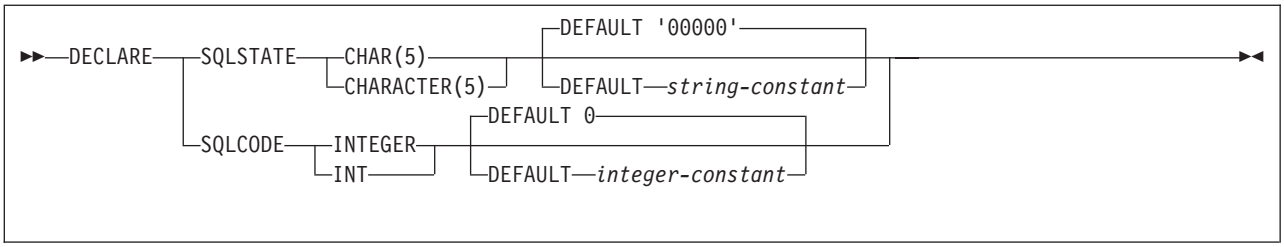
SQL-variable-declaration:



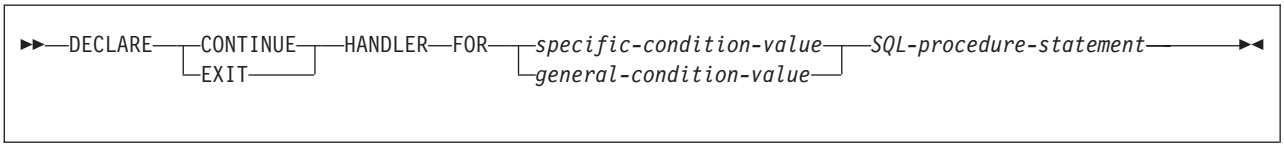
SQL-condition-declaration:



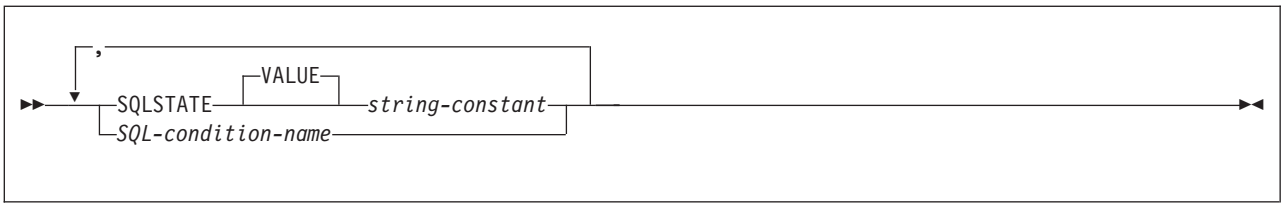
return-codes-declaration:



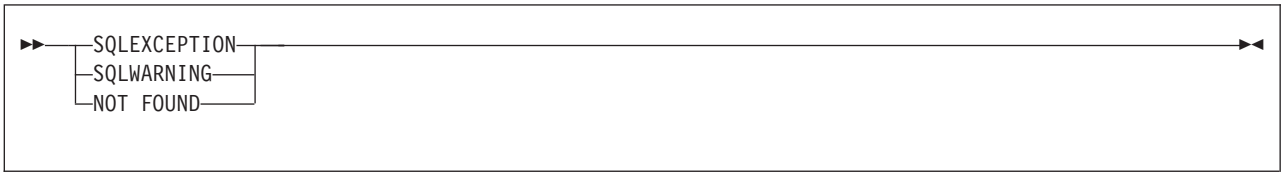
handler-declaration:



specific-condition-value:



general-condition-value:



Description

label
Defines the label for the code block. If the beginning label is specified, it can be used to qualify SQL variables declared in the compound statement and can also be specified on a LEAVE statement. If the ending label is specified, it must be the same as the beginning label.

NOT ATOMIC
NOT ATOMIC indicates that an error within the compound statement does not cause the compound statement to be rolled back.

SQL-variable-declaration
Declares a variable that is local to the compound statement.

SQL-variable-name
A qualified or unqualified name that designates a variable in an SQL procedure body. The unqualified form of *SQL-variable-name* is an SQL identifier and must not be a delimited identifier that contains lowercase letters or special characters. The qualified form is an SQL procedure statement label followed by a period (.) and an SQL identifier.

DB2 folds all SBCS SQL variable names to uppercase. SQL variable names should not be the same as column names. If an SQL statement contains an SQL variable or parameter and a column reference with the same name, DB2 interprets the name as an SQL variable or parameter name. To refer to the column, qualify the column name with the table name. Further, to avoid ambiguous variable references and to ensure compatibility with other DB2 platforms, qualify the SQL variable or parameter name with the label of the SQL procedure statement.

data-type
Specifies the data type and length of the variable. SQL variables follow the same rules for default lengths and maximum lengths as SQL procedure parameters. See "CREATE PROCEDURE (SQL - external)" on page 1338 for a description of SQL data types and lengths.

DEFAULT *constant* **or NULL**
Defines the default for the SQL variable. The variable is initialized when the SQL procedure is called. If a default value is not specified, the variable is initialized to NULL.

RESULT_SET_LOCATOR VARYING
Specifies the data type for a result set locator variable.

SQL-condition-declaration
Declares a condition name and corresponding SQLSTATE value.

SQL-condition-name
Specifies the name of the condition. The condition name is an SQL identifier and must not be a delimited identifier that includes lowercase letters or special characters. *SQL-condition-name* must be unique within the procedure body and can be referenced only within the compound statement in which it is declared.

FOR SQLSTATE *string-constant*
Specifies the SQLSTATE that is associated with the condition. The string must be specified as five characters enclosed in single quotes, and cannot be '00000'.

return-codes-declaration
Declares special variables called SQLSTATE and SQLCODE that are set automatically to the value returned after processing an SQL statement. Both the SQLSTATE and SQLCODE variables can be declared only in the outermost compound statement of the SQL procedure. Assignment to these variables is not prohibited; however, assignment is ignored by exception handlers, and processing the next SQL statement replaces the assigned value.

DECLARE-CURSOR-statement
Declares a cursor. Each cursor in the procedure body must have a unique name. An OPEN statement must be specified to open the cursor, and a FETCH statement can be specified to read rows. The cursor can be referenced only from within the compound statement. For more information on declaring a cursor, see "DECLARE CURSOR" on page 1535.

handler-declaration
Specifies a set of statements to execute when an exception or completion condition occurs in the compound statement. *SQL-procedure-statement* is the set of statements that execute when the handler receives control. See "SQL-procedure-statement" on page 2035 for information on *SQL-procedure-statement*.

A handler is active only within the compound statement in which it is declared.

The actions that a handler can perform are:

CONTINUE
Specifies that after the condition handler is activated and completes successfully, control is returned to the SQL statement that follows the statement that raised the condition. However, if the condition is an error condition and it was encountered while evaluating a search condition, as in a CASE, IF, REPEAT or WHILE statement, control returns to the statement that follows the corresponding END CASE, END IF, END REPEAT, or END WHILE.

EXIT
After the handler is invoked successfully, control is returned to the end of the compound statement.

The conditions that can cause the handler to gain control are:

SQLSTATE *string-constant*
Specifies an SQLSTATE for which the handler is invoked. The SQLSTATE cannot be '00000'.

SQL-condition-name
Specifies a condition name for which the handler is invoked. The condition name must be previously defined in a condition declaration.

SQLEXCEPTION
Specifies that the handler is invoked when an SQLEXCEPTION occurs. An SQLEXCEPTION is an SQLSTATE in which the class code is a value other than '00', '01', or '02'. For more information on SQLSTATE values, see *DB2 Codes*.

SQLWARNING
Specifies that the handler is invoked when an SQLWARNING occurs. An SQLWARNING is an SQLSTATE value with a class code of '01'.

NOT FOUND
Specifies that the handler is invoked when a NOT FOUND condition occurs. NOT FOUND corresponds to an SQLSTATE value with a class code of '02'.

Notes

The order of statements in a compound statement must be:
1. SQL variable, condition declarations, and return codes declarations
2. Cursor declarations
3. Handler declarations
4. SQL procedure statements

Compound statements cannot be nested.

Unlike host variables, SQL variables are not preceded by colons when they are used in SQL statements.

The following rules apply to handlers:

- A handler declaration that contains SQLEXCEPTION, SQLWARNING, or NOT FOUND cannot contain additional SQLSTATE or condition names.

- Handler declarations within the same compound statement cannot contain duplicate conditions.
- A handler declaration cannot contain the same condition code or SQLSTATE value more than once, and cannot contain an SQLSTATE value and a condition name that represent the same SQLSTATE value.
- A handler is activated when it is the most appropriate handler for an exception or completion condition.
- If there is no handler for an SQL error, the error is passed to the caller in the SQLCA.
- A handler cannot be activated by an assignment statement that assigns a value to SQLSTATE.

The following rules and recommendations apply to the SQLCODE and SQLSTATE SQL variables:

- A null value cannot be assigned to SQLSTATE or SQLCODE.
- The SQLSTATE and SQLCODE variable values should be saved immediately to temporary variables if there is any intention to use the values. If a handler exists for SQLSTATE, this assignment must be done as the first statement to be processed in the handler to avoid having the value replaced by the next SQL procedure statement. If the condition raised by the SQL statement is handled, the value is changed by the first SQL statement contained in the handler.

Considerations for the SQLSTATE and SQLCODE SQL variables: The compound statement itself does not affect the SQLSTATE and SQLCODE SQL variables. However, SQL statements contained within the compound statement can affect the SQLSTATE and SQLCODE SQL variables. At the end of the compound statement, the SQLSTATE and SQLCODE SQL variables reflect the result of the last SQL statement executed within the compound statement that caused a change to the SQLSTATE and SQLCODE SQL variables. If the SQLSTATE and SQLCODE SQL variables were not changed within the compound statement, they contain the same values as when the compound statement was entered.

Examples

Create a procedure body with a compound statement that performs the following actions:

- Declares SQL variables, a condition for SQLSTATE '02000', a handler for the condition, and a cursor
- Opens the cursor, fetches a row, and closes the cursor

```
CREATE PROCEDURE PROC1(OUT NOROWS INT) LANGUAGE SQL
BEGIN
 DECLARE v_firstnme VARCHAR(12);
 DECLARE v_midinit CHAR(1);
 DECLARE v_lastname VARCHAR(15);
 DECLARE v_edlevel SMALLINT;
 DECLARE v_salary DECIMAL(9,2);
 DECLARE at_end INT DEFAULT 0;
 DECLARE not_found
  CONDITION FOR '02000';
 DECLARE c1 CURSOR FOR
  SELECT FIRSTNME, MIDINIT, LASTNAME,
   EDLEVEL, SALARY
  FROM EMP;
 DECLARE CONTINUE HANDLER FOR not_found SET NOROWS=1;
```

```
 OPEN c1;
 FETCH c1 INTO v_firstnme, v_midinit,
  v_lastname, v_edlevel, v_salary;
END
```

GET DIAGNOSTICS statement

The GET DIAGNOSTICS statement obtains information about the previous SQL statement that was executed.

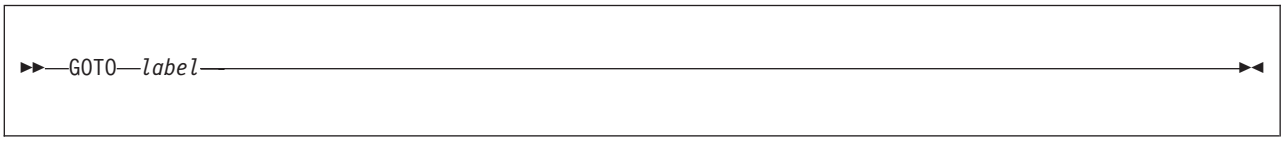
See “GET DIAGNOSTICS” on page 1679.

When you need to specify a variable in a GET DIAGNOSTICS statement that is used within an SQL procedure, you would use either *SQL-variable-name* or *SQL-parameter-name*. In an embedded GET DIAGNOSTICS statement, you would use a *host-variable*. You can replace the instances of *host-variable* in the description of “GET DIAGNOSTICS” on page 1679 with *SQL-variable-name* or *SQL-parameter-name*.

GOTO statement

The GOTO statement is used to branch to a user-defined label within an SQL procedure.

Syntax



Description

label

Specifies a labeled statement at which processing is to continue.

The labeled statement and the GOTO statement must be in the same scope. The following rules apply to the scope:

- If the GOTO statement is defined in a compound statement, *label* must be defined inside the same compound statement.
- If the GOTO statement is defined in a handler, *label* must be defined in the same handler and follow the other scope rules.
- If the GOTO statement is defined outside of a handler, *label* must not be defined within a handler.

If *label* is not defined within a scope that the GOTO statement can reach, an error is returned.

A label name cannot be the same as the name of the SQL procedure in which the label is used.

Notes

Use the GOTO statement sparingly. Because the GOTO statement interferes with the normal sequence of processing, it makes an SQL procedure more difficult to read and maintain. Before using a GOTO statement, determine whether some other statement, such as an IF statement or LEAVE statement, can be used instead.

Examples

Use a GOTO statement to transfer control to the end of a compound statement if the value of an SQL variable is less than 600.

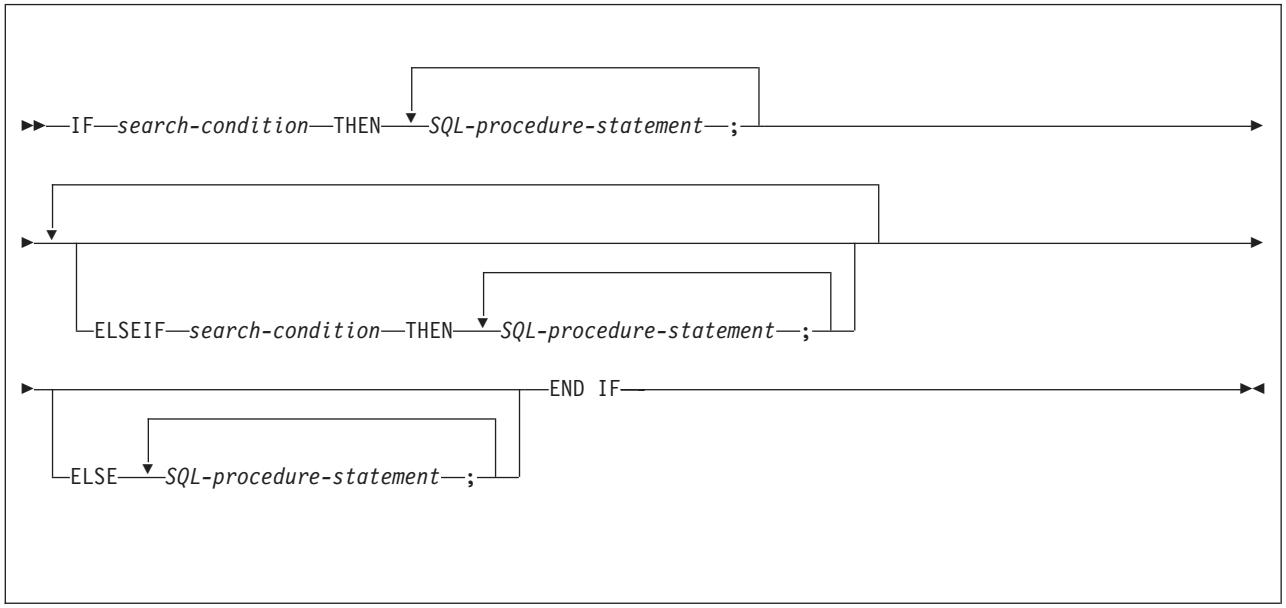
```
BEGIN
 DECLARE new_salary DECIMAL(9,2);
 DECLARE service DECIMAL(8,2);
 SELECT SALARY, CURRENT_DATE - HIREDATE
  INTO new_salary, service
 FROM EMP
WHERE EMPNO = v_empno;
IF service < 600
 THEN GOTO EXIT;
END IF;
IF rating = 1
 THEN SET new_salary =
  new_salary + (new_salary * .10);
ELSEIF rating = 2
 THEN SET new_salary =
  new_salary + (new_salary * .05);
```

```
END IF;
UPDATE EMP
SET SALARY = new_salary
WHERE EMPNO = v_empno;
EXIT: SET return_parm = service;
END
```

IF statement

The IF statement selects an execution path based on the evaluation of a condition.

Syntax



Description

search-condition
Specifies the condition for which an SQL statement should be invoked. If the condition is unknown or false, processing continues to the next search condition until either a condition is true or processing reaches the ELSE clause.

SQL-procedure-statement
Specifies the statement to be invoked if the preceding *search-condition* is true. If no *search-condition* evaluates to true, then the *SQL-procedure-statement* following the ELSE keyword is invoked. The statement must be one of the statements listed under "SQL-procedure-statement" on page 2035.

Notes

Considerations for the SQLSTATE and SQLCODE SQL variables: When the first SQL-procedure-statement in the IF statement is executed, the SQLSTATE and SQLCODE SQL variables reflect the result of evaluating the search conditions of that IF statement. If an IF statement does not include an ELSE clause and none of the search conditions evaluate to true, then when the statement that follows that IF statement is executed, the SQLSTATE and SQLCODE SQL variables reflect the result of evaluating the search conditions of that IF statement.

Examples

Assign a value to the SQL variable *new_salary* based on the value of SQL variable rating.

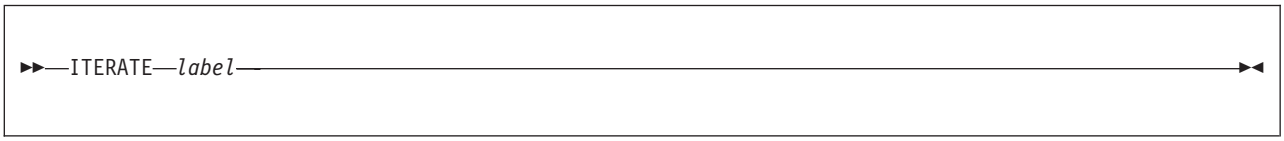
```
IF rating = 1
 THEN SET new_salary =
  new_salary + (new_salary * .10);
 ELSEIF rating = 2
```

```
  THEN SET new_salary =
   new_salary + (new_salary * .05);
 ELSE SET new_salary =
  new_salary + (new_salary * .02);
END IF
```

ITERATE statement

The ITERATE statement causes the flow of control to return to the beginning of a labeled loop.

Syntax



Description

label

Specifies the label of the LOOP, REPEAT, or WHILE statement to which the flow of control is passed.

Examples

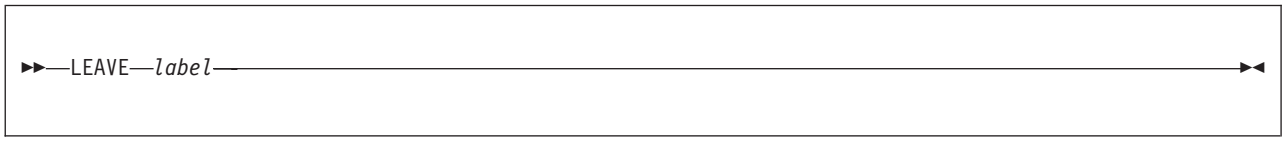
This example uses a cursor to return information for a new department. If the not_found condition handler is invoked, the flow of control passes out of the loop. If the value of *v_dept* is 'D11', an ITERATE statement causes the flow of control to be passed back to the top of the LOOP statement. Otherwise, a new row is inserted into the table.

```
CREATE PROCEDURE ITERATOR ()
  LANGUAGE SQL
  MODIFIES SQL DATA
  BEGIN
    DECLARE v_dept CHAR(3);
    DECLARE v_deptname VARCHAR(29);
    DECLARE v_admdept CHAR(3);
    DECLARE at_end INTEGER DEFAULT 0;
    DECLARE not_found CONDITION FOR SQLSTATE '02000';
    DECLARE c1 CURSOR FOR
      SELECT deptno,deptname,admrdept
        FROM department
        ORDER BY deptno;
    DECLARE CONTINUE HANDLER FOR not_found
    SET at_end = 1;
    OPEN c1;
    ins_loop:
    LOOP
      FETCH c1 INTO v_dept, v_deptname, v_admdept;
      IF at_end = 1 THEN
        LEAVE ins_loop;
      ELSEIF v_dept = 'D11' THEN
        ITERATE ins_loop;
      END IF;
      INSERT INTO department (deptno,deptname,admrdept)
                      VALUES('NEW', v_deptname, v_admdept);
    END LOOP;
    CLOSE c1;
  END
```

LEAVE statement

The LEAVE statement transfers program control out of a loop or a compound statement.

Syntax



Description

label

Specifies the label of the compound statement or loop to exit.

A label name cannot be the same as the name of the SQL procedure in which the label is used.

Notes

When a LEAVE statement transfers control out of a compound statement, all open cursors in the compound statement, except cursors that are used to return result sets, are closed.

Examples

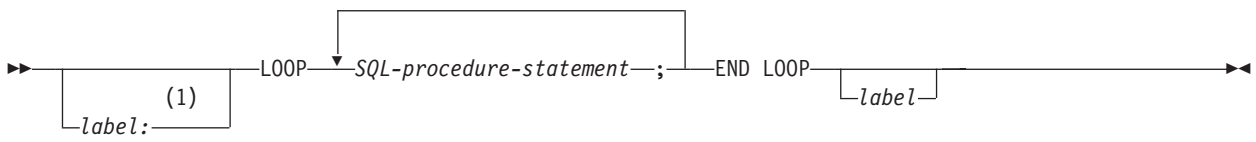
Use a LEAVE statement to transfer control out of a LOOP statement when a negative SQLCODE occurs.

```
ftch_loop: LOOP
 FETCH c1 INTO
  v_firstnme, v_midinit,
  v_lastname, v_edlevel, v_salary;
 IF SQLCODE=100 THEN LEAVE ftch_loop;
 END IF;
END LOOP
```

LOOP statement

The LOOP statement executes a statement or group of statements multiple times.

Syntax



Notes:

1 Only one *label:* can be specified for each *SQL-procedure-statement*.

Description

label

Specifies the label for the LOOP statement. If the ending label is specified, the beginning label must be specified, and the two must match.

A label name cannot be the same as the name of the SQL procedure in which the label is used.

SQL-procedure-statement

Specifies the statements to be executed in the loop. The statement must be one of the statements listed under "SQL-procedure-statement" on page 2035.

Examples

This procedure uses a LOOP statement to fetch values from the employee table. Each time the loop iterates, the OUT parameter counter is incremented and the value of *v_midinit* is checked to ensure that the value is not a single space (' '). If *v_midinit* is a single space, the LEAVE statement passes the flow of control outside of the loop.

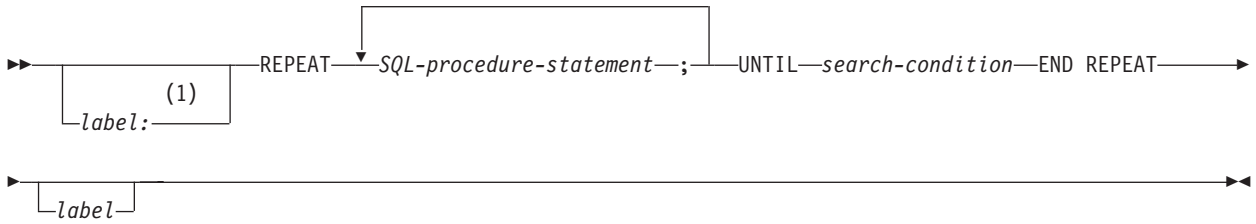
```
CREATE PROCEDURE LOOP_UNTIL_SPACE(OUT counter INTEGER)
   LANGUAGE SQL
   BEGIN
      DECLARE v_counter INTEGER DEFAULT 0;
      DECLARE v_firstnme VARCHAR(12);
      DECLARE v_midinit CHAR(1);
      DECLARE v_lastname VARCHAR(15);
      DECLARE c1 CURSOR FOR
         SELECT firstnme, midinit, lastname
           FROM employee;
      DECLARE EXIT HANDLER FOR NOT FOUND
         SET counter = -1;
      OPEN c1;
      fetch_loop:
      LOOP
        FETCH c1 INTO v_firstnme, v_midinit, v_lastname;
        IF v_midinit = ' ' THEN
           LEAVE fetch_loop;
        END IF;
        SET v_counter = v_counter + 1;
```

```
   END LOOP fetch_loop;
   SET counter = v_counter;
   CLOSE c1;
END
```

REPEAT statement

The REPEAT statement executes a statement or group of statements until a search condition is true.

Syntax



Notes:

1 Only one *label*: can be specified for each *SQL-procedure-statement*.

Description

label
Specifies the label for the REPEAT statement. If the ending label is specified, the beginning label must be specified, and the two must match.

A label name cannot be the same as the name of the SQL procedure in which the label is used.

SQL-procedure-statement
Specifies the statements to be executed. The statement must be one of the statements listed under "SQL-procedure-statement" on page 2035.

search-condition
Specifies a condition that is evaluated after each execution of the REPEAT statement. If the condition is true, the REPEAT loop will exit. If the condition is unknown or false, the REPEAT loop continues.

Examples

Use a REPEAT statement to fetch rows from a table.

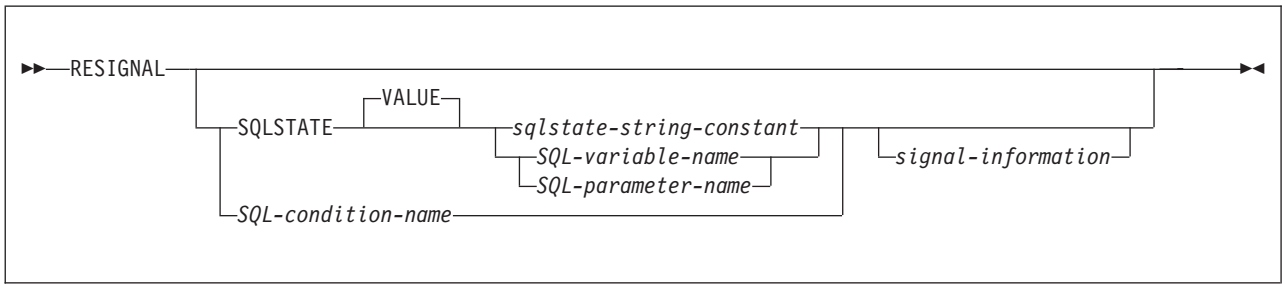
```
fetch_loop:
REPEAT
 FETCH c1 INTO
  v_firstnme, v_midinit, v_lastname;
UNTIL
    SQLCODE <> 0
END REPEAT fetch_loop
```

RESIGNAL statement

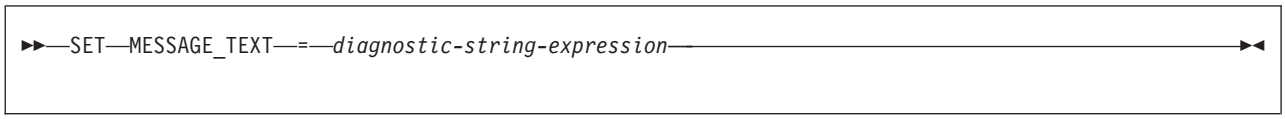
The RESIGNAL statement is used within a condition handler to re-raise the current condition, or to raise an alternate condition so that it can be processed at a higher level. It causes an exception, warning, or not found condition to be returned along with optional message text.

Issuing the RESIGNAL statement without an operand causes the current condition to be passed upwards.

Syntax



signal-information:



Description

SQLSTATE VALUE

Specifies the SQLSTATE that will be returned. Any valid SQLSTATE value can be used. It must be a character string constant with exactly five characters that follow the rules for SQLSTATE values:

- Each character must be from the set of digits ('0' through '9') or non-accented upper case letter ('A' through 'Z').
- The SQLSTATE class (the first two characters) cannot be '00' because it represents successful completion.

If the SQLSTATE does not conform to these rules, an error occurs.

sqlstate-string-constant

A character string constant with a length of five bytes that is a valid SQLSTATE value.

SQL-variable-name **or** *SQL-parameter-name*

Specifies an SQL variable or SQL parameter that is defined for the procedure.

SQL-variable-name

Specifies an SQL variable that is declared within the *compound-statement* that contains the RESIGNAL statement. *SQL-variable-name* must be defined as CHAR or VARCHAR data type with a length of five bytes, must not be null, and must contain a valid SQLSTATE value.

SQL_parameter-name

Specifies an SQL parameter that is defined for the procedure that contains the SQLSTATE value. The SQL parameter must be defined as

a CHAR or VARCHAR value and have a length of five bytes and must not be null. The SQL parameter must contain a valid SQLSTATE value.

SQL-condition-name
Specifies the name of the condition that will be returned. *condition-name* must be declared within the *compound-statement*.

SET MESSAGE_TEXT
Specifies a string that describes the error or warning. The string is returned in the SQLERRMC field of the SQLCA or with the GET DIAGNOSTICS statement.

diagnostic-string-expression
An expression with a data type of CHAR or VARCHAR that returns a character string of up to 1000 bytes that describes the error or warning condition. For information on how to obtain the complete message text, see "GET DIAGNOSTICS" on page 1679.

Notes

While any valid SQLSTATE value can be used in the RESIGNAL statement, programmers should define new SQLSTATE values based on ranges reserved for applications. This practice prevents the unintentional use of an SQLSTATE value that might be defined by the database manager in a future release.

If the RESIGNAL statement is issued without an SQLSTATE clause or a *condition-name*, the RESIGNAL statement must be in a handler and the identical condition that activated the handler is returned. The SQLSTATE, SQLCODE, and the SQLCA associated with the condition are unchanged.

If an SQLSTATE clause or a *condition-name* was specified, the SQLCODE returned is based on the SQLSTATE value as follows:

- If the specified SQLSTATE class is either '01' or '02', a warning or not-found message is returned, and the SQLCODE is set to +438.
- Otherwise, an exception is returned and the SQLCODE is set to -438.

The other fields of the SQLCA are set as follows:

- SQLERRD*x* fields are set to zero.
- SQLWARN*x* fields are set to blank.
- SQLERRMC is set to the first 70 bytes of MESSAGE_TEXT.
- SQLERRML is set to the length of SQLERRMC.
- SQLERRP is set to ROUTINE.

When the SQLSTATE or condition indicates that an exception is returned (an SQLSTATE class other than '01' or '02'), the exception is not handled, and control is immediately returned to the end of the compound statement.

When the SQLSTATE or condition indicates that a warning (SQLSTATE class '02') is returned, the warning is not handled, and processing continues with the next statement.

When the SQLSTATE or condition indicates that a not-found condition (SQLSTATE class '02') is returned, the not-found condition is not handled, and processing continues with the next statement.

Examples

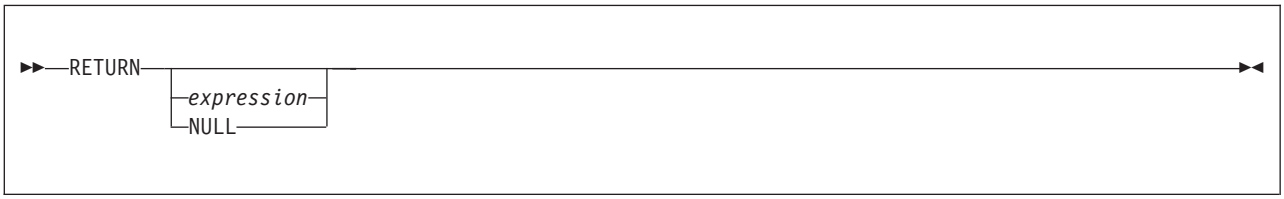
The following example detects a division by zero error. The IF statement uses a SIGNAL statement to invoke the overflow condition handler. The condition handler uses a RESIGNAL statement to return a different SQLSTATE to the client application.

```
CREATE PROCEDURE divide (  IN numerator INTEGER,
                                  IN denominator INTEGER,
                                  OUT divide_result INTEGER)
 LANGUAGE SQL
 CONTAINS SQL
BEGIN
   DECLARE overflow CONDITION for SQLSTATE '22003' ;
     DECLARE CONTINUE HANDLER FOR overflow
     RESIGNAL SQLSTATE '22375';
   IF denominator = 0 THEN
   SIGNAL overflow;
ELSE
  SET divide_result = numerator / denominator;
END IF;
END
```

RETURN statement

The RETURN statement is used to return from the routine. For SQL functions, it returns the result of the function. For an SQL procedure, it optionally returns an integer status value.

Syntax



Description

expression
Specifies a value that is returned from the routine.

- If the routine is a function, *expression* must be specified and the value of *expression* must conform to the SQL assignment rules as described in "Assignment and comparison" on page 121. If the value is being assigned to a string variable, storage assignment rules apply.
- If the routine is a procedure, the data type of *expression* must be INTEGER. If *expression* evaluates to the null value, a value of 0 is returned.

The *expression* cannot include a column name or a host variable. See "Expressions" on page 240 for information on expressions. The *expression* cannot contain a scalar fullselect.

NULL
The null value is returned from the SQL function. **NULL** is not allowed in SQL procedures.

Notes

When a RETURN statement is not used within an SQL procedure or when no value is specified: If a RETURN statement was not used to return from a procedure or if a value is not specified on the RETURN statement, one of the following values is set:

- If the procedure returns with an SQLCODE that is greater or equal to zero, the return status is set to a value of '0'.
- If the procedure returns with an SQLCODE that is less than zero, the return status is set to a value of '-1'.

When a RETURN statement is used within an SQL procedure: If a RETURN statement with a specified return value was used to return from a procedure, the SQLCODE, SQLSTATE, and message length in the SQLCA are initialized to zeros and the message text is set to blanks. An error is not returned to the caller.

When the value is returned: When a value is returned from a procedure, the caller may access the value using one of the following methods:

- The GET DIAGNOSTICS statement to retrieve the RETURN_STATUS when the SQL procedure was called from another SQL procedure.
- The parameter bound for the return value parameter marker in the escape clause CALL syntax (?=CALL...) in a CLI application.

- Directly from the SQLCA returned from processing the CALL of an SQL procedure by retrieving the value of sqlerrd[0]. When the SQLCODE is less than zero, the sqlerrd[0] value is not set. The application should assume a return status value of '-1'.

Examples

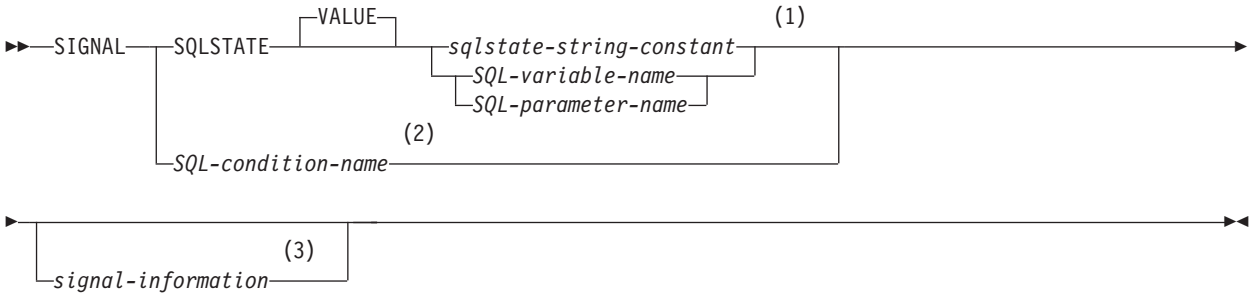
Use a RETURN statement to return from an SQL procedure with a status value of zero if successful or '-200' if not successful.

```
BEGIN
       . . .
           GOTO FAIL;
       . . .
SUCCESS: RETURN 0;
   FAIL: RETURN -200;
END
```

SIGNAL statement

The SIGNAL statement is used to return an error or warning condition. It causes an error or warning to be returned with the specified SQLSTATE, along with optional message text.

Syntax



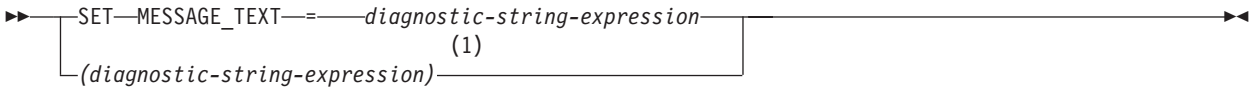
Notes:

1 The **SQLSTATE** variation must be used within a trigger body.

2 *SQL-condition-name* must not be specified within a trigger body.

3 *signal-information* must be specified within a trigger body

signal-information:



Notes:

1 *(diagnostic-string-expression)* must only be specified within a trigger body.

Description

SQLSTATE VALUE

Specifies the SQLSTATE that will be returned. Any valid SQLSTATE value can be used. It must be a character string constant with exactly five characters that follow the rules for SQLSTATEs:

- Each character must be from the set of digits ('0' through '9') or non-accented upper case letter ('A' through 'Z').
- The SQLSTATE class (the first two characters) cannot be '00' because it represents successful completion.

If the SQLSTATE does not conform to these rules, an error occurs.

sqlstate-string-constant

A character string constant with a length of five bytes that is a valid SQLSTATE value.

SQL-variable-name **or** *SQL-parameter-name*
Specifies an SQL variable or SQL parameter that contains a valid SQLSTATE value.

SQL-variable-name
Specifies an SQL variable that is declared within the *compound-statement*. *SQL-variable-name* must be defined as a CHAR or VARCHAR data type, have a length of five bytes, must not be null, and must contain a valid SQLSTATE value.

SQL-parameter-name
Specifies an SQL parameter that is defined for the procedure and contains the SQLSTATE value. The SQL parameter must be defined as a CHAR or VARCHAR value, have a length of five bytes, must not be null, and must contain a valid SQLSTATE value.

SQL-condition-name
Specifies the name of the condition that will be returned. *condition-name* must be declared within the *compound-statement*.

SET MESSAGE_TEXT
Specifies a string that describes the error or warning. The string is returned in the SQLERRMC field of the SQLCA or with the GET DIAGNOSTICS statement.

diagnostic-string-expression
An expression with a data type of CHAR or VARCHAR that returns a character string of up to 1000 bytes that describes the error or warning condition. For information on how to obtain the complete message text, see "GET DIAGNOSTICS" on page 1679.

(diagnostic-string-expression)
An expression with a data type of CHAR or VARCHAR that returns a character string of up to 1000 bytes that describes the error or warning condition. For information on how to obtain the complete message text, see "GET DIAGNOSTICS" on page 1679.

This syntax variation is only provided within the scope of a CREATE TRIGGER statement for compatibility with previous versions of DB2. To conform with the ANS and ISO standards, this form should not be used.

Notes

While any valid SQLSTATE value can be used in the SIGNAL statement, programmers should define new SQLSTATEs based on ranges reserved for applications. This practice prevents the unintentional use of an SQLSTATE value that might be defined by the database manager in a future release.

If a SIGNAL statement is issued, the SQLCODE that is returned is based on the SQLSTATE as follows:

- If the specified SQLSTATE class is either '01' or '02', a warning or not-found message is returned, and the SQLCODE is set to +438.
- Otherwise, an exception is returned and the SQLCODE is set to -438.

The other fields of the SQLCA are set as follows:

- SQLERRDx fields are set to zero.
- SQLWARNx fields are set to blank.
- SQLERRMC is set to the first 70 bytes of MESSAGE_TEXT.

- SQLERRML is set to the length of SQLERRMC.
- SQLERRP is set to ROUTINE.

When the SQLSTATE or condition indicates that an exception (an SQLSTATE class other than '01' or '02') is returned, one of the following actions occurs:

- If a handler exists for the specified SQLSTATE, condition, or SQLEXCEPTION, the exception is handled, and control is transferred to that handler.
- Otherwise, the exception is not handled, and control is immediately returned to the end of the compound statement.

When the SQLSTATE or condition indicates that a warning (SQLSTATE class '01') is returned, one of the following actions occurs:

- If an active handler exists for the specified SQLSTATE, condition, or SQLWARNING, the warning is handled, and control is transferred to that handler.
- Otherwise, the warning is not handled, and processing continues with the next statement.

When the SQLSTATE or condition indicates that a not-found condition (SQLSTATE class '02') is returned, one of the following actions occurs:

- If an active handler exists for the specified SQLSTATE, condition, or not-found condition, the not-found condition is handled, and control is transferred to that handler.
- Otherwise, the not-found condition is not handled, and processing continues with the next statement.

When the SIGNAL statement is issued in a handler, no active handler exists.

Using a SIGNAL statement in the body of a trigger: Within the triggered action of a CREATE TRIGGER statement, the message text can be specified using only these variations:

```
SIGNAL SQLSTATE sqlstate-string-constant
    SET MESSAGE_TEXT = diagnostic-string-expression
SIGNAL SQLSTATE sqlstate-string-constant
   (diagnostic-string-expression)
```

Examples

Example 1: The following example shows an SQL procedure for an order system that signals an application error when a customer number is not known to the application. The ORDERS table includes a foreign key to the CUSTOMER table, requiring that the CUSTNO exist before an order can be inserted.

```
CREATE PROCEDURE SUBMIT_ORDER
            (IN ONUM INTEGER, IN CNUM INTEGER,
             IN PNUM INTEGER, IN QNUM INTEGER)
 LANGUAGE SQL
 SPECIFIC SUBMIT_ORDER
 MODIFIES SQL DATA
BEGIN
   DECLARE EXIT HANDLER FOR SQLSTATE VALUE '23503'
     SIGNAL SQLSTATE '75002'
         SET MESSAGE_TEXT = 'Customer number is not known';
   INSERT INTO ORDERS (ORDERNO, CUSTNO, PARTNO, QUANTITY)
      VALUES (ONUM, CNUM, PNUM, QNUM);
END
```

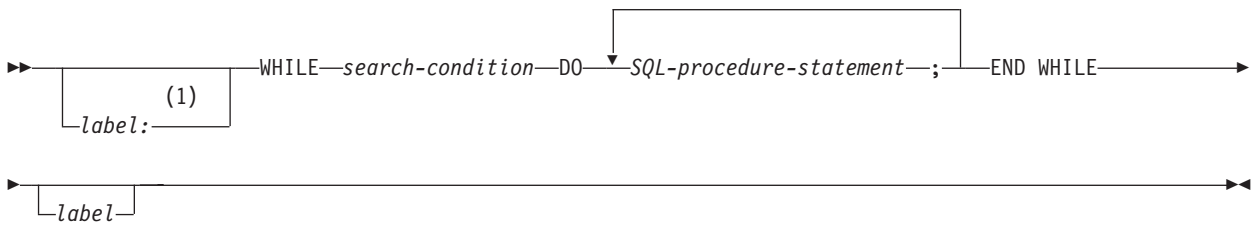
Example 2: The following example shows a trigger for an order system that allows orders to be recorded in an ORDERS table (ORDERNO, CUSTNO, PARTNO, QUANTITY) only if there is sufficient stock in the PARTS tables. When there is insufficient stock for an order, SQLSTATE '75001' is returned along with an appropriate error description.

```
CREATE TRIGGER CK_AVAIL
      NO CASCADE BEFORE INSERT ON ORDERS
      REFERENCING NEW AS NEW_ORDER
      FOR EACH ROW MODE DB2SQL
      WHEN (NEW_ORDER.QUANTITY > (SELECT ON_HAND FROM PARTS
                                  WHERE NEW_ORDER.PARTNO = PARTS.PARTNO))
        BEGIN ATOMIC
          SIGNAL SQLSTATE '75001' ('Insufficient stock for order');
        END
```

WHILE statement

The WHILE statement repeats the execution of a statement or group of statements while a specified condition is true.

Syntax



Notes:

1 Only one *label:* can be specified for each *SQL-procedure-statement*. If an ending label is specified for this beginning label, the labels must be the same.

Description

label
Specifies the label for the WHILE statement. If the ending label is specified, it must be the same as the beginning label.

A label name cannot be the same as the name of the SQL procedure in which the label is used.

search-condition
Specifies a condition that is evaluated before each execution of the loop. If the condition is true, the SQL procedure statement in the loop is executed.

SQL-procedure-statement
Specifies the statements to be executed in the loop. The statement must be one of the statements listed under "SQL-procedure-statement" on page 2035.

Examples

Use a WHILE statement to fetch rows from a table while SQL variable at_end, which indicates whether the end of the table has been reached, is 0.

```
WHILE at_end = 0 DO
 FETCH c1 INTO
  v_firstnme, v_midinit,
  v_lastname, v_edlevel, v_salary;
 IF SQLCODE=100 THEN SET at_end=1;
 END IF;
END WHILE
```

SQL communication area (SQLCA)

An SQLCA is a structure or collection of variables that is updated after each SQL statement executes. An application program that contains executable SQL statements must provide exactly one SQLCA, with a few exceptions.

The following exceptions exist:

- A program that is precompiled with the STDSQL(YES) option must not provide an SQLCA
- In some cases a Fortran program must provide more than one SQLCA.

In all host languages except REXX, the SQL INCLUDE statement can be used to provide the declaration of the SQLCA.

In COBOL and assembler:
The name of the storage area must be SQLCA.

In PL/I, and C:
The name of the structure must be SQLCA. Every executable SQL statement must be within the scope of its declaration.

Unless noted otherwise, *C* is used to represent C/370™ and C/C++ programming languages.

In Fortran:
The name of the COMMON area for the INTEGER variables of the SQLCA must be SQLCA1; the name of the COMMON area for the CHARACTER variables must be SQLCA2. An SQLCA definition is required for every subprogram that contains SQL statements. One is also needed for the main program if it contains SQL statements.

In Java:
The DB2Sqlca class, which is an encapsulation of the SQLCA, should be used.

In REXX:
DB2 generates the SQLCA automatically. A REXX procedure cannot use the INCLUDE statement. The REXX SQLCA has a somewhat different format from SQLCAs for the other languages.

Related reference:

DB2Sqlca class (DB2 Application Programming for Java)

"The REXX SQLCA" on page 2077

Description of SQLCA fields

For the most part, COBOL, C, PL/I, and assembler use the same names for the SQLCA fields, and Fortran uses different names. However, there is one instance where C, PL/I, and assembler names differ from COBOL.

The names in the following table are those provided by the SQL INCLUDE statement.

Table 163. Fields of SQLCA

assembler, COBOL, or PL/I Name	C Name	Fortran Name	Data type	Purpose
SQLCAID	sqlcaid	Not used.	CHAR(8)	An "eye catcher" for storage dumps, containing the text 'SQLCA'. The sixth byte is 'L' if line number information is returned from parsing a dynamic statement or a native SQL procedure. The sixth byte is not set when processing an external SQL procedure.
SQLCABC	sqlcabc	Not used.	INTEGER	Contains the length of the SQLCA: 136.
SQLCODE (See note 1)	SQLCODE	SQLCOD	INTEGER	Contains the SQL return code. (See note 2) **Code** **Means** **0** Successful execution (though there might have been warning messages). **positive** Successful execution, but with a warning condition or other information. **negative** Error condition.
SQLERRML (See note 3)	sqlerrml (See note 3)	SQLTXL	SMALLINT	Length indicator for SQLERRMC, in the range 0 through 70. 0 means that the value of SQLERRMC is not pertinent.
SQLERRMC (See note 3)	sqlerrmc (See note 3)	SQLTXT	VARCHAR(70)	Contains one or more tokens, separated by X'FF', that are substituted for variables in the descriptions of error conditions. It may contain truncated tokens. A message length of 70 bytes indicates a possible truncation.
SQLERRP	sqlerrp	SQLERP	CHAR(8)	Provides a product signature and, in the case of an error, diagnostic information such as the name of the module that detected the error. In all cases, the first three characters are 'DSN' for DB2 for z/OS.
SQLERRD(1)	sqlerrd[0]	SQLERR(1)	INTEGER	For a sensitive static cursor, contains the number of rows in a result table when the cursor position is after the last row (that is, when SQLCODE is equal to +100). On successful return from an SQL procedure, contains the return status value from the SQL procedure. SQLERRD(1) can also contain an internal error code.

Table 163. Fields of SQLCA (continued)

assembler, COBOL, or PL/I Name	C Name	Fortran Name	Data type	Purpose
SQLERRD(2)	sqlerrd[1]	SQLERR(2)	INTEGER	For a sensitive static cursor, contains the number of rows in a result table when the cursor position is after the last row (that is, when SQLCODE is equal to +100). SQLERRD(2) can also contain an internal error code.

Table 163. Fields of SQLCA (continued)

assembler, COBOL, or PL/I Name	C Name	Fortran Name	Data type	Purpose
SQLERRD(3)	sqlerrd[2]	SQLERR(3)	INTEGER	Contains the number of rows that qualified to be deleted, inserted, or updated after a DELETE, INSERT, UPDATE, or MERGE statement. The number excludes rows affected by triggers, referential integrity constraints, or inserted rows that are the result of processing a FOR PORTION OF clause for a BUSINESS_TIME period. For the OPEN of a cursor for a SELECT with a data change statement or for a SELECT INTO, SQLERRD(3) contains the number of rows affected by the embedded data change statement. The value is 0 if the SQL statement fails, indicating that all changes made in executing the statement canceled. For a DELETE statement the value will be -1 if the operation is a mass delete from a table in a segmented table space and the DELETE statement did not include selection criteria. If the delete was against a view, neither the DELETE statement nor the definition of the view included selection criteria. For a TRUNCATE statement, the value will be -1. For a PREPARE statement, contains the estimated number of rows selected. If the number of rows is greater than 2 147 483 647, a value of 2 147 483 647 is returned. For a REFRESH TABLE statement, SQLERRD(3) contains the number of rows inserted into the materialized query table. For a rowset-oriented FETCH, contains the number of rows fetched. For SQLCODES -911 and -913, SQLERRD(3) contains the reason code for the timeout or deadlock. When an error is encountered in parsing a dynamic statement, or when parsing, binding, or executing a native SQL procedure, SQLERRD(3) will contain the line number where the error was encountered. The sixth byte of SQLCAID must be 'L' for this to be a valid line number. This value will be meaningful only if the statement source contains new line control characters. This information is not returned for an external SQL procedure.

Table 163. Fields of SQLCA (continued)

assembler, COBOL, or PL/I Name	C Name	Fortran Name	Data type	Purpose
SQLERRD(4)	sqlerrd[3]	SQLERR(4)	INTEGER	Generally, contains timerons, a short floating-point value that indicates a rough relative estimate of resources required (See note 4). It does not reflect an estimate of the time required. When preparing a dynamically defined SQL statement, you can use this field as an indicator of the relative cost of the prepared SQL statement. For a particular statement, this number can vary with changes to the statistics in the catalog. It is also subject to change between releases of DB2 for z/OS.
SQLERRD(5)	sqlerrd[4]	SQLERR(5)	INTEGER	Contains the position or column of a syntax error for a PREPARE or EXECUTE IMMEDIATE statement.
SQLERRD(6)	sqlerrd[5]	SQLERR(6)	INTEGER	Contains an internal error code.
SQLWARN0	SQLWARN0	SQLWRN(0)	CHAR(1)	Contains a blank if no other indicator is set to a warning condition (that is, no other indicator contains a W or Z). Contains a W if at least one other indicator contains a W or Z.
SQLWARN1	SQLWARN1	SQLWRN(1)	CHAR(1)	Contains a W if the value of a string column was truncated when assigned to a host variable. Contains an N for non-scrollable cursors and S for scrollable cursors after the OPEN CURSOR or ALLOCATE CURSOR statement.
SQLWARN2	SQLWARN2	SQLWRN(2)	CHAR(1)	Contains a W if null values were eliminated from the argument of an aggregate function; not necessarily set to W for the MIN function because its results are not dependent on the elimination of null values.
SQLWARN3	SQLWARN3	SQLWRN(3)	CHAR(1)	Contains a W if the number of result columns is larger than the number of host variables. Contains a Z if fewer locators were provided in the ASSOCIATE LOCATORS statement than the stored procedure returned.
SQLWARN4	SQLWARN4	SQLWRN(4)	CHAR(1)	Contains a W if a prepared UPDATE or DELETE statement does not include a WHERE clause. For a scrollable cursor, contains a D for sensitive dynamic cursors, I for insensitive cursors, and S for sensitive static cursors after the OPEN CURSOR or ALLOCATE CURSOR statement; blank if cursor is not scrollable.
SQLWARN5	SQLWARN5	SQLWRN(5)	CHAR(1)	Contains a W if the SQL statement was not executed because it is not a valid SQL statement in DB2 for z/OS. Contains a character value of 1 (read only), 2 (read and delete), or 4 (read, delete, and update) to reflect capability of the cursor after the OPEN CURSOR or ALLOCATE CURSOR statement.

Table 163. Fields of SQLCA (continued)

assembler, COBOL, or PL/I Name	C Name	Fortran Name	Data type	Purpose
SQLWARN6	SQLWARN6	SQLWRN(6)	CHAR(1)	Contains a W if the addition of a month or year duration to a DATE or TIMESTAMP value results in an invalid day (for example, June 31). Indicates that the value of the day was changed to the last day of the month to make the result valid.
SQLWARN7	SQLWARN7	SQLWRN(7)	CHAR(1)	Contains a W if one or more nonzero digits were eliminated from the fractional part of a number used as the operand of a decimal multiply or divide operation.
SQLWARN8	SQLWARN8	SQLWRX(1)	CHAR(1)	Contains a W if a character that could not be converted was replaced with a substitute character. Contains a Y if there was an unsuccessful attempt to establish a trusted connection.
SQLWARN9	SQLWARN9	SQLWRX(2)	CHAR(1)	Contains a W if arithmetic exceptions were ignored during COUNT or COUNT_BIG processing. Contains a Z if the stored procedure returned multiple result sets.
SQLWARNA	SQLWARNA	SQLWRX(3)	CHAR(1)	Contains a W if at least one character field of the SQLCA or the SQLDA names or labels is invalid due to a character conversion error.
SQLSTATE	sqlstate	SQLSTT	CHAR(5)	Contains a return code for the outcome of the most recent execution of an SQL statement (See note 5).

Notes:

1. With the precompiler option STDSQL(YES) in effect, SQLCODE is replaced by SQLCADE in SQLCA.
2. For the specific meanings of SQL return codes, see *DB2 Codes*.
3. In COBOL, SQLERRM includes SQLERRML and SQLERRMC. In PL/I and C, the varying-length string SQLERRM is equivalent to SQLERRML prefixed to SQLERRMC. In assembler, the storage area SQLERRM is equivalent to SQLERRML and SQLERRMC. See the examples for the various host languages in "The included SQLCA" on page 2075.
4. The use of timerons may require special handling because they are floating-point values in an INTEGER array. In PL/I, for example, you could first copy the value into a BIN FIXED(31) based variable that coincides with a BIN FLOAT(24) variable.
5. For a description of SQLSTATE values, see *DB2 Codes*.

The included SQLCA

The description of the SQLCA that is given by INCLUDE SQLCA is shown for each of the host languages.

assembler:

```
SQLCA    DS   0F
SQLCAID  DS   CL8        ID
SQLCABC  DS   F          BYTE COUNT
SQLCODE  DS   F          RETURN CODE
SQLERRM  DS   H,CL70     ERR MSG PARMS
SQLERRP  DS   CL8        IMPL-DEPENDENT
SQLERRD  DS   6F
SQLWARN  DS   0C         WARNING FLAGS
SQLWARN0 DS   C'W' IF ANY
SQLWARN1 DS   C'W' = WARNING
SQLWARN2 DS   C'W' = WARNING
SQLWARN3 DS   C'W' = WARNING
SQLWARN4 DS   C'W' = WARNING
SQLWARN5 DS   C'W' = WARNING
SQLWARN6 DS   C'W' = WARNING
SQLWARN7 DS   C'W' = WARNING
SQLEXT   DS   0CL8
SQLWARN8 DS   C
SQLWARN9 DS   C
SQLWARNA DS   C
SQLSTATE DS   CL5
```

C:

```
#ifndef SQLCODE
struct sqlca
{
       unsigned char  sqlcaid[8];
       long           sqlcabc;
       long           sqlcode;
       short          sqlerrml;
       unsigned char  sqlerrmc[70];
       unsigned char  sqlerrp[8];
       long           sqlerrd[6];
       unsigned char  sqlwarn[11];
       unsigned char  sqlstate[5];
};
#define  SQLCODE   sqlca.sqlcode
#define  SQLWARN0  sqlca.sqlwarn[0]
#define  SQLWARN1  sqlca.sqlwarn[1]
#define  SQLWARN2  sqlca.sqlwarn[2]
#define  SQLWARN3  sqlca.sqlwarn[3]
#define  SQLWARN4  sqlca.sqlwarn[4]
#define  SQLWARN5  sqlca.sqlwarn[5]
#define  SQLWARN6  sqlca.sqlwarn[6]
#define  SQLWARN7  sqlca.sqlwarn[7]
#define  SQLWARN8  sqlca.sqlwarn[8]
#define  SQLWARN9  sqlca.sqlwarn[9]
#define  SQLWARNA  sqlca.sqlwarn[10]
#define  SQLSTATE  sqlca.sqlstate
#endif
struct sqlca sqlca;
```

COBOL:

```
01 SQLCA.
   05 SQLCAID      PIC X(8).
   05 SQLCABC      PIC S9(9) COMP-5.
   05 SQLCODE      PIC S9(9) COMP-5.
   05 SQLERRM.
      49 SQLERRML  PIC S9(4) COMP-5.
```

```
      49 SQLERRMC  PIC X(70).
   05 SQLERRP      PIC X(8).
   05 SQLERRD      OCCURS 6 TIMES
                   PIC S9(9) COMP-5.
   05 SQLWARN.
      10 SQLWARN0  PIC X.
      10 SQLWARN1  PIC X.
      10 SQLWARN2  PIC X.
      10 SQLWARN3  PIC X.
      10 SQLWARN4  PIC X.
      10 SQLWARN5  PIC X.
      10 SQLWARN6  PIC X.
      10 SQLWARN7  PIC X.
   05 SQLEXT.
      10 SQLWARN8  PIC X.
      10 SQLWARN9  PIC X.
      10 SQLWARNA  PIC X.
      10 SQLSTATE  PIC X(5).
```

Fortran:

```
*
*     THE SQL COMMUNICATIONS AREA
*
      INTEGER     SQLCOD,
     C            SQLERR(6),
     C            SQLTXL*2
      COMMON /SQLCA1/SQLCOD, SQLERR,SQLTXL
      CHARACTER        SQLERP*8,
     C                 SQLWRN(0:7)*1,
     C                 SQLTXT*70,
     C                 SQLEXT*8,
     C                 SQLWRX(1:3)*1,
     C                 SQLSTT*5
      COMMON /SQLCA2/SQLERP,SQLWRN,SQLTXT,SQLWRX,
     C              SQLSTT
      EQUIVALENCE (SQLWRX,SQLEXT)
*
```

PL/I:

```
DECLARE
  1 SQLCA,
    2 SQLCAID CHAR(8),
    2 SQLCABC FIXED(31) BINARY,
    2 SQLCODE FIXED(31) BINARY,
    2 SQLERRM CHAR(70) VAR,
    2 SQLERRP CHAR(8),
    2 SQLERRD(6) FIXED(31) BINARY,
    2 SQLWARN,
      3 SQLWARN0 CHAR(1),
      3 SQLWARN1 CHAR(1),
      3 SQLWARN2 CHAR(1),
      3 SQLWARN3 CHAR(1),
      3 SQLWARN4 CHAR(1),
      3 SQLWARN5 CHAR(1),
      3 SQLWARN6 CHAR(1),
      3 SQLWARN7 CHAR(1),
    2 SQLEXT,
      3 SQLWARN8 CHAR(1),
      3 SQLWARN9 CHAR(1),
      3 SQLWARNA CHAR(1),
      3 SQLSTATE CHAR(5);
```

The REXX SQLCA

The REXX SQLCA consists of a set of variables, rather than a structure. DB2 makes the SQLCA available to your application automatically.

The following table lists the variables in a REXX SQLCA.

Table 164. Variables in a REXX SQLCA

Variable	Contents
SQLCODE	Contains the SQL return code.
SQLERRMC	Contains one or more tokens, separated by X'FF', that are substituted for variables in the descriptions of error conditions. It might contain truncated tokens. A message length of 70 bytes indicates a possible truncation.
SQLERRP	Provides a product signature and, in the case of an error, diagnostic information such as the name of the module that detected the error. For DB2 for z/OS, the product signature is 'DSN'.
SQLERRD.1	For a sensitive static cursor, contains the number of rows in a result table when the cursor position is after the last row (that is, when SQLCODE is equal to +100). SQLERRD(1) can also contain an internal error code.
SQLERRD.2	For a sensitive static cursor, contains the number of rows in a result table when the cursor position is after the last row (that is, when SQLCODE is equal to +100). SQLERRD(2) can also contain an internal error code.
SQLERRD.3	Contains the number of rows that qualified for the operation after an SQL data change statement (but not rows deleted as a result of CASCADE delete). For the OPEN of a cursor for a SELECT with an SQL data change statement or for a SELECT INTO, SQLERRD(3) contains the number of rows affected by the embedded data change statement. Set to 0 if the SQL statement fails, indicating that all changes made in executing the statement were canceled. Set to -1 for a mass delete from a table in a segmented table space, for a truncate operation, or a delete from a view when neither the DELETE statement nor the definition of the view included selection criteria. For rowset-oriented FETCH statements, contains the number of rows returned in the rowset. For SQLCODES -911 and -913, SQLERRD(3) contains the reason code for the timeout or deadlock. After successful execution of the REFRESH TABLE statement, SQLERRD(3) contains the number of rows inserted into the materialized query table. When an error is encountered in parsing a dynamic statement, or when parsing, binding, or executing a native SQL procedure, SQLERRD(3) will contain the line number where the error was encountered. The sixth byte of SQLCAID must be 'L' for this to be a valid line number. This value will be meaningful only if the statement source contains new line control characters. This information is not returned for an external SQL procedure.
SQLERRD.4	Generally, contains timerons, a short floating-point value that indicates a rough relative estimate of resources required. This value does not reflect an estimate of the time required to execute the SQL statement. After you prepare an SQL statement, you can use this field as an indicator of the relative cost of the prepared SQL statement. For a particular statement, this number can vary with changes to the statistics in the catalog. This value is subject to change between releases of DB2 for z/OS.
SQLERRD.5	Contains the position or column of a syntax error for a PREPARE or EXECUTE IMMEDIATE statement.
SQLERRD.6	Contains an internal error code.

Table 164. Variables in a REXX SQLCA (continued)

Variable	Contents
SQLWARN.0	Contains a blank if no other indicator is set to a warning condition (that is, no other indicator contains a W or Z). Contains a W if at least one other indicator contains a W or Z.
SQLWARN.1	Contains a W if the value of a string column was truncated when assigned to a host variable. Contains an N for non-scrollable cursors and S for scrollable cursors after the OPEN CURSOR or ALLOCATE CURSOR statement.
SQLWARN.2	Contains a W if null values were eliminated from the argument of an aggregate function; not necessarily set to W for the MIN function because its results are not dependent on the elimination of null values.
SQLWARN.3	Contains a W if the number of result columns is larger than the number of host variables. Contains Z if the ASSOCIATE LOCATORS statement contains fewer locators than the stored procedure returned.
SQLWARN.4	Contains a W if a prepared UPDATE or DELETE statement does not include a WHERE clause. For a scrollable cursor, contains a D for sensitive dynamic cursors, I for insensitive cursors, and S for sensitive static cursors after the OPEN CURSOR or ALLOCATE CURSOR statement; otherwise, blank if cursor is not scrollable.
SQLWARN.5	Contains a W if the SQL statement was not executed because it is not a valid SQL statement in DB2 for z/OS. Contains a character value of 1 (read only), 2 (read and delete), or 4 (read, delete, and update) to reflect capability of the cursor after the OPEN CURSOR or ALLOCATE CURSOR statement.
SQLWARN.6	Contains a W if the addition of a month or year duration to a DATE or TIMESTAMP value results in an invalid day (for example, June 31). Indicates that the value of the day was changed to the last day of the month to make the result valid.
SQLWARN.7	Contains a W if one or more nonzero digits were eliminated from the fractional part of a number that was used as the operand of a decimal multiply or divide operation.
SQLWARN.8	Contains a W if a character that could not be converted was replaced with a substitute character.
SQLWARN.9	Contains a W if arithmetic exceptions were ignored during COUNT or COUNT_BIG processing. Contains a Z if the stored procedure returned multiple result sets.
SQLWARN.10	Contains a W if at least one character field of the SQLCA is invalid due to a character conversion error.
SQLSTATE	Contains a return code for the outcome of the most recent execution of an SQL statement.

SQL descriptor area (SQLDA)

An SQLDA is a collection of variables that is required for execution of the SQL DESCRIBE statement, and can be optionally used by the PREPARE, OPEN, FETCH, EXECUTE, and CALL statements. An SQLDA can be used in a DESCRIBE or PREPARE INTO statement, modified with the addresses of host variables, and then reused in a FETCH statement.

The meaning of the information in an SQLDA depends on the context in which it is used. For DESCRIBE and PREPARE INTO, DB2 sets the fields in the SQLDA to provide information to the application program. For OPEN, EXECUTE, FETCH, and CALL, the application program sets the fields in the SQLDA to provide DB2 with information:

DESCRIBE *statement-name* **or PREPARE INTO**
With the exception of SQLN, DB2 sets fields of the SQLDA to provide information to an application program about a prepared statement. Each SQLVAR occurrence describes a column of the result table.

DESCRIBE TABLE
With the exception of SQLN, DB2 sets fields of the SQLDA to provide information to an application program about the columns of a table or view. Each SQLVAR occurrence describes a column of the specified table or view.

DESCRIBE CURSOR
With the exception of SQLN, DB2 sets fields of the SQLDA to provide information to an application program about the result set that is associated with the specified cursor. Each SQLVAR occurrence describes a column of the result set.

DESCRIBE INPUT
With the exception of SQLN, DB2 sets fields of the SQLDA to provide information to an application program about the input parameter markers of a prepared statement. Each SQLVAR occurrence describes an input parameter marker.

DESCRIBE PROCEDURE
With the exception of SQLN, DB2 sets fields of the SQLDA to provide information to an application program about the result sets returned by the specified stored procedure. Each SQLVAR occurrence describes a returned result set.

OPEN, EXECUTE, FETCH, and CALL
The application program sets fields of the SQLDA to provide information about host variables or output buffers in the application program to DB2. Each SQLVAR occurrence describes a host variable or output buffer.

- For OPEN and EXECUTE, each SQLVAR occurrence describes an input value that is substituted for a parameter marker in the associated SQL statement that was previously prepared.
- For FETCH, each SQLVAR occurrence describes a host variable or buffer in the application program that is to be used to contain an output value from a row of the result.
- For CALL, each SQLVAR occurrence describes a host variable that corresponds to a parameter in the parameter list for the stored procedure.

For information on the way to use the SQLDA, see *DB2 Application Programming and SQL Guide*.

Description of SQLDA fields

An SQLDA consists of four variables, a header, and an arbitrary number of occurrences of a sequence of variables collectively named SQLVAR.

In DESCRIBE and PREPARE INTO, each occurrence of the SQLVAR describes the column of a table. In FETCH, OPEN, EXECUTE, and CALL, each occurrence describes a host variable.

The order of the SQLVAR entries matches the order of the columns in the table or the order of the parameter markers in the SQL statement.

The SQLDA Header

The fields in the SQLDA header have different usage depending on whether the SQLDA is being used in a DESCRIBE or PREPARE INTO statement or the SQLDA is being used in a FETCH, INSERT, OPEN, EXECUTE, or CALL statement.

The following table describes the fields in the SQLDA header.

Table 165. Fields of the SQLDA header

C name assembler, COBOL or PL/I name	Data type	Usage in DESCRIBE[1] and PREPARE INTO	Usage in FETCH,INSERT, OPEN, EXECUTE, and CALL
sqldaid SQLDAID	CHAR(8)	An "eye catcher" for storage dumps, containing the text 'SQLDA '. The 7th byte of the field is a flag that can be used to determine if more than one SQLVAR entry is needed for each column. For details, see "Determining how many SQLVAR occurrences are needed" on page 2084. For DESCRIBE CURSOR, the field is set to 'SQLRS'. If the cursor is declared WITH HOLD in a stored procedure, the high-order bit of the 8th byte is set to 1. For DESCRIBE PROCEDURE, it is set to 'SQLPR'.	A plus sign (+) in the 6th byte indicates that SQLNAME contains an overriding CCSID. A '2' in the 7th byte indicates the two SQLVAR entries were allocated for each column or parameter. A '3' in the 7th byte indicates that three SQLVAR entries were allocated for each column or parameter. Although three entries are never needed on input to DB2, an SQLDA with three entries might be used when the SQLDA was initialized by a DESCRIBE or PREPARE INTO with a USING BOTH clause. Otherwise, SQLDAID is not used.
sqldabc SQLDABC	INTEGER	Length of the SQLDA, equal to `SQLNx * 44+16`.	Length of the SQLDA, greater than or equal to `SQLNx * 44+16`.

Table 165. Fields of the SQLDA header (continued)

C name assembler, COBOL or PL/I name	Data type	Usage in DESCRIBE[1] and PREPARE INTO	Usage in FETCH,INSERT, OPEN, EXECUTE, and CALL
sqln SQLN	SMALLINT	Unchanged by DB2. The field must be set to a value greater than or equal to zero before the statement is executed. The field indicates the total number of occurrences of SQLVAR. At the very least, the number should be: • For DESCRIBE INPUT, the number of parameter markers to be described. • For other DESCRIBEs or PREPARE INTO: the number of columns of the result, or a multiple of the columns of the result when there are multiple sets of SQLVAR entries because column labels are returned in addition to column names.	Total number of occurrences of SQLVAR provided in the SQLDA. SQLN must be set to a value greater than or equal to zero.
sqld SQLD	SMALLINT	The number of columns described by occurrences of SQLVAR. Double that number if USING BOTH appears in the DESCRIBE or PREPARE INTO statement. Contains a 0 if the statement string is not a query. For DESCRIBE PROCEDURE, the number of result sets returned by the stored procedure. Contains a 0 if no result sets are returned.	The number of host variables described by occurrences of SQLVAR.

Note:

1. The third column of this table represents several forms of the DESCRIBE statement:
   - For DESCRIBE *output* and PREPARE INTO, the column pertains to columns of the result table.
   - For DESCRIBE CURSOR, the column pertains to a result set associated with a cursor.
   - For DESCRIBE INPUT, the column pertains to parameter markers.
   - For DESCRIBE PROCEDURE, the column pertains to the result sets returned by the stored procedure.

SQLVAR entries

For each column or host variable described by the SQLDA, there are both base SQLVAR entries and extended SQLVAR entries.

Base SQLVAR entry

The base SQLVAR entry is always present. The fields of this entry contain the base information about the column or host variable such as data type code, length attribute (except for LOBs), column name (or label), host variable address, and indicator variable address.

Extended SQLVAR entry

The extended SQLVAR entry is needed (for each column) if the result

includes any LOB or distinct type[38] columns. For distinct types, the extended SQLVAR contains the distinct type name. For LOBs, the extended SQLVAR contains the length attribute of the host variable and a pointer to the buffer that contains the actual length. If locators are used to represent LOBs, an extended SQLVAR is not necessary.

The extended SQLVAR entry is also needed for each column when the USING BOTH clause was specified, which indicates that both columns names and labels are returned. (**DESCRIBE** *output* is the only statement with the **USING BOTH** clause).

The fields in the extended SQLVAR that return LOB and distinct type information do not overlap, and the fields that return LOB and label information do not overlap. Depending on the combination of labels, LOBs and distinct types, more than one extended SQLVAR entry per column may be required to return the information. See "Determining how many SQLVAR occurrences are needed" on page 2084.

The following table shows how to map the base and extended SQLVAR entries. For an SQLDA that contains both base and extended SQLVAR entries, the base SQLVAR entries are in the first block, followed by a block of extended SQLVAR entries, which if necessary, are followed by a second block of extended SQLVAR entries. In each block, the number of occurrences of the SQLVAR entry is equal to the value in SQLD[39] even though many of the extended SQLVAR entries might be unused.

Table 166. Contents of SQLVAR arrays

					Set of SQLVAR entries		
LOBs	**Distinct types[1]**	**7th byte of SQLDAID**	**SQLD**	**Minimum for SQLN[2]**	**First set (Base)**	**Second set (Extended)**	**Third set (Extended)**
USING BOTH clause not specified:							
No	No	Space	*n*	*n*	Column names, labels	Not Used	Not Used
Yes[3]	Yes[3]	2	*n*	*2n*	Column names, labels	LOBs, distinct types, or both	Not used
USING BOTH clause was specified:							
No	No	Space	*2n*	*2n*	Column names	Labels	Not used
Yes	No	2	*n*	*2n*	Column names	LOBs and labels	Not used
No	Yes	3	*n*	*3n*	Column names	Distinct types	Labels
Yes	Yes	3	*n*	*3n*	Column names	LOBs and distinct types	Labels

38. DESCRIBE INPUT does not return information about distinct types.

39. When an extended SQLVAR entry is present for each column for *labels* (and there are no LOB or distinct type columns in the result),

Table 166. Contents of SQLVAR arrays (continued)

					Set of SQLVAR entries		
LOBs	**Distinct types[1]**	**7th byte of SQLDAID**	**SQLD**	**Minimum for SQLN[2]**	**First set (Base)**	**Second set (Extended)**	**Third set (Extended)**

Notes:

1. DESCRIBE INPUT does not return information about distinct types.
2. The number of columns or host variables that the SQLDA describes.
3. Either LOBs, distinct types, or both are present.
4. Here, the 7th byte is set to a space and SQLD is set to two times the number of columns in the result. For all other values of the 7th byte for USING BOTH, SQLD is set to the number of columns in the result, and the 7th byte can be used to determine how many SQLVAR entries are needed for each column of the result.

Determining how many SQLVAR occurrences are needed:

The number of SQLVAR occurrences needed depends on the statement that the SQLDA was provided for and the data types of the columns or parameters being described.

If the USING BOTH clause was not specified for the statement and neither LOBs nor distinct types are present in the result, only one SQLVAR entry (a base entry) is needed for each column. The 7th byte of SQLDAID is set to a space. The SQLD is set to the number of columns in the result and represents the number of SQLVAR occurrences needed. If an insufficient number of SQLVAR occurrences were provided, DB2 returns a +236 warning in SQLCODE if the standards option was set. Otherwise, SQLCODE is zero.

If USING BOTH is specified and neither LOBs nor distinct types are present in the result, an extended SQLVAR entry per column is needed for the labels in addition to the base SQLVAR entry. The 7th byte of the SQLDAID is set to space. SQLD is set to the twice the number of columns in the result and represents the combined number of base and extended SQLVAR occurrences needed.

If LOBs, distinct types, or both are present in the results, one extended SQLVAR entry is needed per column in addition to the base SQLVAR entry with one exception. The exception is that when the USING BOTH clause is specified and distinct types are present in the results, two extended SQLVAR entries per column are needed. When a sufficient number of SQLVAR entries are provided in the SQLDA for both the base and extended SQLVARs, information for the LOBs and distinct types is returned. The 7th byte of SQLDAID is set to the number of SQLVAR entries that were used for each column:

2 Two SQLVAR entries per column (a base and an extended)

3 Three SQLVAR entries per column (a base and two extended)

SQLD is set to the number of columns in the result. Therefore, the value of the 7th byte of SQLDAID multiplied by the value of SQLD is the total number SQLVAR entries that were provided.

Otherwise, when an insufficient number of SQLVAR entries have been provided when LOBs or distinct types are present, DB2 indicates that by returning one of the following warnings in SQLCODE. DB2 also sets the 7th byte of SQLDAID to indicate how many SQLVAR entries are needed for each column of the result.

+237 There are insufficient SQLVAR entries to describe the data, and the data

includes distinct types. In this case, there were enough base SQLVAR entries to describe the data, so the base SQLVAR entries are set. However, sufficient extended SQLVAR entries were not provided so the distinct type names are not returned.

+238 There are insufficient SQLVAR entries to describe the data, and the data includes LOBs. In this case no information is returned in the SQLVAR entries.

+239 There are insufficient SQLVAR entries to describe the data, and the data includes distinct types. There weren't even enough base SQLVAR entries. In this case no information is returned in the SQLVAR entries.

Field descriptions of an occurrence of a base SQLVAR:

The fields of a base SQLVAR have different uses depending on the SQL statement.

The following table describes the contents of the fields of a base SQLVAR.

Table 167. Fields in an occurrence of a base SQLVAR

C name assembler COBOL, or PL/I name	Data type	Usage in DESCRIBE[1] and PREPARE INTO	Usage in FETCH, OPEN, EXECUTE, and CALL
sqltype SQLTYPE	SMALLINT	Indicates the data type of the column or parameter and whether it can contain null values. For a description of the type codes, see Table 169 on page 2090. For a distinct type, the data type on which the distinct type was based is placed in this field. The base SQLVAR provides no indication that this is part of the description of a distinct type. For VARCHAR columns defined with CCSID 1208 in an EBCDIC table, SQLTYPE reflects VARCHAR, even though the column is recorded in the catalog as VARBINARY with CCSID 1208. For VARGRAPHIC columns defined with CCSID 1200 in an EBCDIC table, SQLTYPE reflects VARGRAPHIC, even though the column is recorded in the catalog as VARBINARY with CCSID 1200.	Indicates the data type of the host variable and whether an indicator variable is provided. Host variables for datetime values must be character string variables. For FETCH, a datetime type code means a fixed-length character string. For a description of the type codes, see "SQLTYPE and SQLLEN" on page 2090.

Table 167. Fields in an occurrence of a base SQLVAR (continued)

C name assembler COBOL, or PL/I name	Data type	Usage in DESCRIBE[1] and PREPARE INTO	Usage in FETCH, OPEN, EXECUTE, and CALL
sqllen SQLLEN	SMALLINT	The length attribute of the column or parameter. For datetime data, the length of the string representation of the value. See "SQLTYPE and SQLLEN" on page 2090 for a description of allowable values. For LOBs, the value is 0 regardless of the length attribute of the LOB. For XML, the value is 0. Field SQLLONGLEN in the extended SQLVAR contains the length attribute.	The length attribute of the host variable. See "SQLTYPE and SQLLEN" on page 2090 for a description of allowable values. For LOBs, the value is 0 regardless of the length attribute of the LOB. Field SQLLONGLEN in the extended SQLVAR contains the length attribute. For XML AS BLOB, CLOB, or DBCLOB, sqllen is 0 as for LOB types.
sqldata SQLDATA	pointer	For string columns or parameters, SQLDATA contains `X'0000zzzz'`, where *zzzz* is the associated CCSID. For character strings, SQLDATA can alternatively contain X'FFFF', which indicates bit data. Not used for other types of data. For datetime columns, SQLDATA can contain the CCSID of the string representation of the datetime value. For DESCRIBE PROCEDURE, the result set locator value associated with the result set. For VARCHAR columns defined with CCSID 1208 in an EBCDIC table SQLDATA contains 1208, even though the column might be recorded in the catalog as VARBINARY with CCSID 1208. For VARGRAPHIC columns defined with CCSID 1200 in an EBCDIC table SQLTYPE contains 1200, even though the column might be recorded in the catalog as VARBINARY with CCSID 1200.	Contains the address of the host variable.
sqlind SQLIND	pointer	Reserved For DESCRIBE PROCEDURE, it is set to -1.	Contains the address of an associated indicator variable, if SQLTYPE is odd. Otherwise, the field is not used.

Table 167. Fields in an occurrence of a base SQLVAR (continued)

C name assembler COBOL, or PL/I name	Data type	Usage in DESCRIBE[1] and PREPARE INTO	Usage in FETCH, OPEN, EXECUTE, and CALL
sqlname SQLNAME	VARCHAR(30)	Contains the unqualified name or label of the column, or a string of length zero if the name or label does not exist. If the name is longer than 30 bytes, it is truncated at a byte boundary. For more information about column names, see Names of result columns. For DESCRIBE PROCEDURE, SQLNAME contains the cursor name used by the stored procedure to return the result set. The values for SQLNAME appear in the order the cursors were opened by the stored procedure. For DESCRIBE INPUT, SQLNAME is not used.	Can contain CCSID and/or host-variable-array dimension information. DB2 interprets the third and fourth byte of the data portion of SQLNAME as the CCSID of the host variable if all of the following are true and the third and fourth byte are not X'0000': • The 6th byte of SQLDAID is '+' (x'4E') • SQLTYPE indicates the host variable is a string variable • The length of SQLNAME is 8 • The first two bytes of the data portion of SQLNAME are X'0000'. If the third and fourth byte of the data portion of SQLNAME are X'0000', DB2 uses the appropriate default CCSID. For FETCH, OPEN, INSERT, and EXECUTE, if the length of SQLNAME is 8, and the first two bytes of the data portion of SQLNAME are X'0000', DB2 interprets the fifth through eighth bytes of the data portion of the SQLNAME field as follows:

Table 167. Fields in an occurrence of a base SQLVAR (continued)

C name assembler COBOL, or PL/I name	Data type	Usage in DESCRIBE[1] and PREPARE INTO	Usage in FETCH, OPEN, EXECUTE, and CALL
(cont.) sqlname SQLNAME			• **fifth and sixth bytes:** a flag field that indicates the type of host variable that is being described by the current SQLDA entry. The values of this field are as follows: – X'0000' - host variable – X'0100' - XML host variable (XML AS BLOB, XML AS CLOB, XML AS DBCLOB) – X'0001' - host variable array – X'0101' - XML host variable array – X'0002' - special host variable that represents the value for 'n' in a multiple-row INSERT statement. • **seventh and eighth bytes:** if the sixth byte is X'01', a binary small integer (halfword) that represents the dimension of the host-variable-array, and the corresponding indicator-array if one is specified.

Notes:

1. The third column of this table represents several forms of the DESCRIBE statement.
   - For DESCRIBE *output* and PREPARE INTO, the column pertains to columns of the result table.
   - For DESCRIBE CURSOR, the column pertains to a result set associated with a cursor.
   - For DESCRIBE INPUT, the column pertains to parameter markers.
   - For DESCRIBE PROCEDURE, the column pertains to the result sets returned by the stored procedure.

Field descriptions of an occurrence of an extended SQLVAR:

The fields of an extended SQLVAR have different uses depending on the SQL statement.

The following table describes the contents of the fields of an extended SQLVAR entry.

Table 168. Fields in an occurrence of an extended SQLVAR

C name assembler, COBOL, or PL/I name	Data type	Usage in DESCRIBE[1] and PREPARE INTO	Usage in FETCH, OPEN, EXECUTE, and CALL
len.sqllonglen SQLLONGL SQLLONGLEN	INTEGER	The length attribute of a LOB (BLOB, CLOB, or DBCLOB) column.	The length attribute of a LOB (BLOB, CLOB, or DBCLOB) host variable. DB2 ignores the SQLLEN field in the base SQLVAR for these data types. The length attribute indicates the number of bytes for a BLOB or CLOB, and the number of characters for a DBCLOB.
*	INTEGER	Reserved.	Reserved.
sqldatalen SQLDATAL SQLDATALEN	pointer	Not used.	Used only for LOB (BLOB, CLOB, and DBCLOB) host variables. If the value of the field is null, the actual length of the LOB is stored in the 4 bytes immediately before the start of the data, and SQLDATA points to the first byte of the field length. The actual length indicates the number of bytes for a BLOB or CLOB, and the number of characters for a DBCLOB. If the value of the field is not null, the field contains a pointer to a 4-byte long buffer that contains the actual length *in bytes* (even for DBCLOBs) of the data in the buffer pointed to from the SQLDATA field in the matching base SQLVAR. Regardless of whether this field is used, field SQLLONGLEN must be set.

Table 168. Fields in an occurrence of an extended SQLVAR (continued)

C name assembler, COBOL, or PL/I name	Data type	Usage in DESCRIBE[1] and PREPARE INTO	Usage in FETCH, OPEN, EXECUTE, and CALL
sqldatatype_name SQLTNAME SQLDATATYPE-NAME	VARCHAR(30)	A SQLTNAME field of the extended SQLVAR is set to one of the following: • For a distinct type column, the database manager sets this field to the fully qualified distinct type name. If the qualified name is longer than 30 bytes, it is truncated. • For a label, the database manager sets this field to label. In the case that both a distinct type name and a label need to be returned in extended SQLVAR entries (USING BOTH has been specified), the distinct type name is returned in the first extended SQLVAR entry and the label in the second extended SQLVAR entry.	Not used.

Note:

1. The third column of this table represents several forms of the DESCRIBE statement.
   - For DESCRIBE *output* and PREPARE INTO, the column pertains to columns of the result table.
   - For DESCRIBE CURSOR, the column pertains to a result set associated with a cursor.
   - For DESCRIBE INPUT, the column pertains to parameter markers.
   - For DESCRIBE PROCEDURE, the column pertains to the result sets returned by the stored procedure.

SQLTYPE and SQLLEN:

The contents of the SQLTYPE and SQLLEN fields of the SQLDA depends on the SQL statement that is returning the value.

The following table shows the values that can appear in the SQLTYPE and SQLLEN fields of the SQLDA. In DESCRIBE and PREPARE INTO, an even value of SQLTYPE means the column does not allow nulls, and an odd value means the column does allow nulls. In DESCRIBE INPUT statements, only odd values are returned for SQLTYPE. In FETCH, OPEN, EXECUTE, and CALL, an even value of SQLTYPE means no indicator variable is provided, and an odd value means that SQLIND contains the address of an indicator variable.

Table 169. SQLTYPE and SQLLEN values for DESCRIBE, PREPARE INTO, FETCH, OPEN, EXECUTE, and CALL

	For DESCRIBE and PREPARE INTO		For FETCH, OPEN, EXECUTE, and CALL	
SQLTYPE	Column or parameter data type	SQLLEN	Host variable data type	SQLLEN
384/385	date	10 [1]	fixed-length character string representation of a date	length attribute of the host variable

Table 169. SQLTYPE and SQLLEN values for DESCRIBE, PREPARE INTO, FETCH, OPEN, EXECUTE, and CALL (continued)

	For DESCRIBE and PREPARE INTO		For FETCH, OPEN, EXECUTE, and CALL	
SQLTYPE	**Column or parameter data type**	**SQLLEN**	**Host variable data type**	**SQLLEN**
388/389	time	8 [2]	fixed-length character string representation of a time	length attribute of the host variable
392/393	timestamp without time zone	19 for TIMESTAMP(0) otherwise 20+*p* for TIMESTAMP(*p*) [5]	fixed-length character string representation of a timestamp	length attribute of the host variable
400/401	N/A	N/A	NUL-terminated graphic string	length attribute of the host variable
404/405	BLOB	0 [3]	BLOB or XML AS BLOB	Not used. [3]
408/409	CLOB	0 [3]	CLOB or XML AS CLOB	Not used. [3]
412/413	DBCLOB	0 [3]	DBCLOB or XML AS DBCLOB	Not used. [3]
448/449	varying-length character string	length attribute of the column	varying-length character string	length attribute of the host variable
452/453	fixed-length character string	length attribute of the column	fixed-length character string	length attribute of the host variable
456/457	long varying-length character string SQLTYPE values 448/449 are returned instead of 456/457 for statements that are bound in Version 9 or later.	length attribute of the column	long varying-length character string	length attribute of the host variable
460/461	N/A	N/A	NUL-terminated character string	length attribute of the host variable
464/465	varying-length graphic string	length attribute of the column	varying-length graphic string	length attribute of the host variable
468/469	fixed-length graphic string	length attribute of the column	fixed-length graphic string	length attribute of the host variable
472/473	long graphic string SQLTYPE values 464/465 are returned instead of 472/473 for statements that are bound in Version 9 or later.	length attribute of the column	long graphic string	length attribute of the host variable
480/481	floating point	4 for single precision, 8 for double precision	floating point	4 for single precision, 8 for double precision
484/485	packed decimal	precision in byte 1; scale in byte 2	packed decimal	precision in byte 1; scale in byte 2
492/493	big integer[4]	8	big integer	8
496/497	large integer	4	large integer	4
500/501	small integer	2	small integer	2

Table 169. SQLTYPE and SQLLEN values for DESCRIBE, PREPARE INTO, FETCH, OPEN, EXECUTE, and CALL (continued)

	For DESCRIBE and PREPARE INTO		For FETCH, OPEN, EXECUTE, and CALL	
SQLTYPE	**Column or parameter data type**	**SQLLEN**	**Host variable data type**	**SQLLEN**
504/505	N/A	N/A	DISPLAY SIGN LEADING SEPARATE, NATIONAL SIGN LEADING SEPARATE	precision in byte 1; scale in byte 2
904/905	ROWID	40	ROWID	40
908/909	varying-length binary string	length attribute of the column	varying-length binary string	length attribute of the host variable
912/913	fixed-length binary string	length attribute of the column	fixed-length binary string	length attribute of the host variable
916/917	BLOB_FILE	267		
920/921	CLOB_FILE	267		
924/925	DBCLOB_FILE	267		
960/961	BLOB locator	4	BLOB_LOCATOR	4
964/965	CLOB locator	4	CLOB_LOCATOR	4
968/969	DBCLOB locator	4	DBCLOB_LOCATOR	4
988/989	XML	0	Invalid. Instead, use one of the following: XML AS BLOB, XML AS CLOB, XML AS DBCLOB	Not used
996/997	DECFLOAT(16) DECFLOAT(34)	8 16	DECFLOAT(16) DECFLOAT(34)	8 16
2448/2449	timestamp with time zone	147 for TIMESTAMP(0) WITH TIME ZONE otherwise 148+*p* for TIMESTAMP(*p*) WITH TIME ZONE [5]	varying-length character string representation of a timestamp with time zone	length attribute of the host variable

Note:

1. SQLLEN might be different if a date installation exit is specified.
2. SQLLEN might be different if a time installation exit is specified.
3. Field SQLLONGLEN in the extended SQLVAR contains the length attribute of the column.
4. BIGINT is supported by other DB2 platforms.
5. *p* is the timestamp precision.

SQLDATA:

Depending on the data type of the string column that the SQLVAR is describing, the SQLDATA field can contain different CCSID values.

The following table identifies the CCSID values that appear in the SQLDATA field when the SQLVAR element describes a string column.

Table 170. CCSID values for SQLDATA

Data type	Subtype	Bytes 1 and 2	Bytes 3 and 4
Character	SBCS data	X'0000'	CCSID
Character	mixed data	X'0000'	CCSID
Character	BIT data	X'0000'	X'FFFF'
Graphic	N/A	X'0000'	CCSID
Any other data type	N/A	N/A	N/A

Unrecognized and unsupported SQLTYPES

The values that appear in the SQLTYPE field of the SQLDA are dependent on the level of data type support available at the sender as well as at the receiver of the data. This support is particularly important as new data types are added to the product.

New data types might not be supported by the sender or receiver of the data and might not be recognized by the sender or receiver of the data. Depending on the situation, the new data type might be returned, a compatible data type that is agreed to by both the sender and the receiver of the data might be returned, or an error might occur.

When the sender and receiver agree to use a compatible data type, the following table indicates the mapping that takes place. This mapping takes place when at least one of the sender or receiver does not support the data type provided. The unsupported data type can be provided by either the application or the database manager.

Table 171. Compatible data types for unsupported data types

Data type	Compatible data type
ROWID	VARCHAR(40) FOR BIT DATA

No indication is given in the SQLDA that the data type is substituted.

The included SQLDA

Only assembler, C, C++, COBOL, and PL/I C are supported for the SQLDA that is given by INCLUDE SQLDA.

assembler:

```
SQLTRIPL EQU    C'3'
SQLDOUBL EQU    C'2'
SQLSINGL EQU    C' '
*
         SQLSECT SAVE
*
SQLDA    DSECT
SQLDAID  DS    CL8      ID
SQLDABC  DS    F        BYTE COUNT
SQLN     DS    H        COUNT SQLVAR/SQLVAR2 ENTRIES
SQLD     DS    H        COUNT VARS (TWICE IF USING BOTH)
*
SQLVAR   DS    0F       BEGIN VARS
SQLVARN  DSECT ,        NTH VARIABLE
SQLTYPE  DS    H        DATA TYPE CODE
SQLLEN   DS    0H       LENGTH
SQLPRCSN DS    X        DEC PRECISION
SQLSCALE DS    X        DEC SCALE
SQLDATA  DS    A        ADDR OF VAR
SQLIND   DS    A        ADDR OF IND
SQLNAME  DS    H,CL30   DESCRIBE NAME
SQLVSIZ  EQU   *-SQLDATA
SQLSIZV  EQU   *-SQLVARN
*
SQLDA    DSECT
SQLVAR2  DS     0F      BEGIN EXTENDED FIELDS OF VARS
SQLVAR2N DSECT  ,       EXTENDED FIELDS OF NTH VARIABLE
SQLLONGL DS     F       LENGTH
SQLRSVDL DS     F       RESERVED
SQLDATAL DS     A       ADDR OF LENGTH IN BYTES
SQLTNAME DS     H,CL30  DESCRIBE NAME
*
         SQLSECT RESTORE
```

In the above declaration, SQLSECT SAVE is a macro invocation that remembers the current CSECT name. SQLSECT RESTORE is a macro invocation that continues that CSECT.

C and C++:

```
#ifndef  SQLDASIZE                /* Permit duplicate Includes      */
 /**/
 struct sqlvar
        { short  sqltype;
          short  sqllen;
          char  *sqldata;
          short *sqlind;
          struct sqlname
                { short  length;
                  char   data[30];
                } sqlname;
        };
 /**/
 struct sqlvar2
        { struct
                { long   sqllonglen;
         unsigned long   reserved;
                } len;
          char  *sqldatalen;
          struct sqldistinct_type
```

```
               { short  length;
                 char   data[30];
               } sqldatatype_name;
       };
 /**/
 struct sqlda
       { char   sqldaid[8];
         long   sqldabc;
         short  sqln;
         short  sqld;
         struct sqlvar sqlvar[1];
   };
 /**/
/*********************************************************************/
/* Macros for using the sqlvar2 fields.                              */
/*********************************************************************/
 /**/
/*********************************************************************/
/*   '2' in the 7th byte of sqldaid indicates a doubled number of    */
/*       sqlvar entries.                                             */
/*   '3' in the 7th byte of sqldaid indicates a tripled number of    */
/*       sqlvar entries.                                             */
/*********************************************************************/
#define   SQLDOUBLED  '2'
#define   SQLTRIPLED  '3'
#define   SQLSINGLED  ' '
 /**/
/*********************************************************************/
/* GETSQLDOUBLED(daptr) returns 1 if the SQLDA pointed to by         */
/* daptr has been doubled, or 0 if it has not been doubled.          */
/*********************************************************************/
#define GETSQLDOUBLED(daptr) \
    (((daptr)->sqldaid[6] == ( char) SQLDOUBLED) ? \
    (1)         : \
    (0)           )
 /**/
/*********************************************************************/
/* GETSQLTRIPLED(daptr) returns 1 if the SQLDA pointed to by         */
/* daptr has been tripled, or 0 if it has not been tripled.          */
/*********************************************************************/
#define GETSQLTRIPLED(daptr) \
    (((daptr)->sqldaid[6] == ( char) SQLTRIPLED) ? \
    (1)         : \
    (0)           )
 /**/
/*********************************************************************/
/* SETSQLDOUBLED(daptr, SQLDOUBLED) sets the 7th byte of sqldaid     */
/* to '2'.                                                           */
/* SETSQLDOUBLED(daptr, SQLSINGLED) sets the 7th byte of sqldaid     */
/* to be a ' '.                                                      */
/*********************************************************************/
#define SETSQLDOUBLED(daptr, newvalue) \
    (((daptr)->sqldaid[6] = (newvalue)))
 /**/
/*********************************************************************/
/* SETSQLTRIPLED(daptr) sets the 7th byte of sqldaid                 */
/* to '3'.                                                           */
/*********************************************************************/
#define SETSQLTRIPLED(daptr) \
    (((daptr)->sqldaid[6] = (SQLTRIPLED)))
 /**/
/*********************************************************************/
/* GETSQLDALONGLEN(daptr,n) returns the data length of the nth       */
/* entry in the sqlda pointed to by daptr. Use this only if the      */
/* sqlda was doubled or tripled and the nth SQLVAR entry has a       */
/* LOB datatype.                                                     */
/*********************************************************************/
```

```
#define GETSQLDALONGLEN(daptr,n)    (    \
    (long) (((struct sqlvar2 *) &((daptr);->sqlvar[(n) + \
           ((daptr)->sqld)])) \
            ->len.sqllonglen))
 /**/
/*********************************************************************/
/* SETSQLDALONGLEN(daptr,n,len) sets the sqllonglen field of the     */
/* sqlda pointed to by daptr to len for the nth entry. Use this only */
/* if the sqlda was doubled or tripled and the nth SQLVAR entry has  */
/* a LOB datatype.                                                   */
/*********************************************************************/
#define SETSQLDALONGLEN(daptr,n,length)  { \
    struct sqlvar2     *var2ptr; \
    var2ptr = (struct sqlvar2 *) \
        &((daptr);->sqlvar[(n) + ((daptr)->sqld)]); \
    var2ptr->len.sqllonglen  =  (long ) (length); \
    }
 /**/
/*********************************************************************/
/* GETSQLDALENPTR(daptr,n) returns a pointer to the data length for  */
/* the nth entry in the sqlda pointed to by daptr. Unlike the inline */
/* value (union sql8bytelen len), which is 8 bytes, the sqldatalen   */
/* pointer field returns a pointer to a long (4 byte) integer.       */
/* If the SQLDATALEN pointer is zero, a NULL pointer is be returned. */
/*                                                                   */
/* NOTE: Use this only if the sqlda has been doubled or tripled.     */
/*********************************************************************/
#define GETSQLDALENPTR(daptr,n) (   \
    (((struct sqlvar2 *) &(daptr);->sqlvar[(n) + (daptr)->sqld]) \
                          ->sqldatalen == NULL) ? \
    ((long *) NULL ) : \
    ((long *) ((struct sqlvar2 *) \
        &(daptr);->sqlvar[(n) + (daptr)->sqld]) \
                                    ->sqldatalen ) )
 /**/
/*********************************************************************/
/* SETSQLDALENPTR(daptr,n,ptr) sets a pointer to the data length for */
/* the nth entry in the sqlda pointed to by daptr.                   */
/* Use this only if the sqlda has been doubled or tripled.           */
/*********************************************************************/
#define SETSQLDALENPTR(daptr,n,ptr)  {  \
    struct sqlvar2 *var2ptr;    \
    var2ptr = (struct sqlvar2 *) \
        &((daptr);->sqlvar[(n) + ((daptr)->sqld)]); \
    var2ptr->sqldatalen  = (char *) ptr; \
    }
 /**/
#define SQLDASIZE(n) \
   ( sizeof(struct sqlda) + ((n)-1) * sizeof(struct sqlvar) )
#endif /* SQLDASIZE */
```

COBOL (IBM COBOL only):

```
01 SQLDA.
   05 SQLDAID PIC X(8).
   05 SQLDABC PIC S9(9) BINARY.
   05 SQLN    PIC S9(4) BINARY.
   05 SQLD    PIC S9(4) BINARY.
   05 SQLVAR  OCCURS 0 TO 750 TIMES DEPENDING ON SQLN.
      10 SQLVAR1.
         15 SQLTYPE PIC S9(4) BINARY.
         15 SQLLEN  PIC S9(4) BINARY.
         15 FILLER  REDEFINES SQLLEN.
            20 SQLPRECISION   PIC X.
            20 SQLSCALE       PIC X.
         15 SQLDATA POINTER.
         15 SQLIND  POINTER.
```

```
         15 SQLNAME.
            49 SQLNAMEL PIC S9(4) BINARY.
            49 SQLNAMEC PIC X(30).
      10 SQLVAR2 REDEFINES SQLVAR1.
         15 SQLVAR2-RESERVED-1 PIC S9(9) BINARY.
         15 SQLLONGLEN         REDEFINES SQLVAR2-RESERVED-1
                               PIC S9(9) BINARY.
         15 SQLVAR2-RESERVED-2 PIC S9(9) BINARY.
         15 SQLDATALEN         POINTER.
         15 SQLDATATYPE-NAME.
             20 SQLDATATYPE-NAMEL PIC S9(4) BINARY.
             20 SQLDATATYPE-NAMEC PIC X(30).
```

PL/I:

```
DECLARE
  1 SQLDA BASED(SQLDAPTR),
    2 SQLDAID CHAR(8),
    2 SQLDABC FIXED(31) BINARY,
    2 SQLN    FIXED(15) BINARY,
    2 SQLD    FIXED(15) BINARY,
    2 SQLVAR(SQLSIZE REFER(SQLN)),
      3 SQLTYPE  FIXED(15) BINARY,
      3 SQLLEN   FIXED(15) BINARY,
      3 SQLDATA  POINTER,
      3 SQLIND   POINTER,
      3 SQLNAME  CHAR(30)  VAR;
 /* */
DECLARE
  1 SQLDA2 BASED(SQLDAPTR),
    2 SQLDAID2 CHAR(8),
    2 SQLDABC2 FIXED(31) BINARY,
    2 SQLN2    FIXED(15) BINARY,
    2 SQLD2    FIXED(15) BINARY,
    2 SQLVAR2(SQLSIZE REFER(SQLN2)),
      3 SQLBIGLEN,
        4 SQLLONGL FIXED(31) BINARY,
        4 SQLRSVDL FIXED(31) BINARY,
      3 SQLDATAL POINTER,
      3 SQLTNAME CHAR(30) VAR;
 /* */
DECLARE SQLSIZE    FIXED(15) BINARY;
DECLARE SQLDAPTR   POINTER;
DECLARE SQLTRIPLED CHAR(1)   INITIAL('3');
DECLARE SQLDOUBLED CHAR(1)   INITIAL('2');
DECLARE SQLSINGLED CHAR(1)   INITIAL(' ');
```

Identifying an SQLDA in C or C++

A *descriptor-name* can be specified in the CALL, DESCRIBE, EXECUTE, FETCH, and OPEN statements. When the host application is written in C or C++, *descriptor-name* can be a pointer variable with pointer notation.

For example, *descriptor-name* could be declared as

```
sqlda *outsqlda;
```

Afterwords, it could be used in a statement like the following:

```
EXEC SQL DESCRIBE STMT1 INTO DESCRIPTOR :*outsqlda;
```

The REXX SQLDA

A REXX SQLDA consists of a set of REXX variables with a common stem. The stem must be a REXX variable name that contains no periods and is the same as the value of *descriptor-name* that you specify when you use the SQLDA in an SQL statement. DB2 does not support the INCLUDE SQLDA statement in REXX.

The following table shows the variable names in a REXX SQLDA. The values in the second column of the table are values that DB2 inserts into the SQLDA when the statement executes. Except where noted otherwise, the values in the third column of the table are values that the application must put in the SQLDA before the statement executes.

Table 172. Fields of a REXX SQLDA

Variable name	Usage in DESCRIBE and PREPARE INTO	Usage in FETCH, OPEN, EXECUTE, and CALL
stem.SQLD	The number of columns that are described in the SQLDA. Double that number if USING BOTH appears in the DESCRIBE or PREPARE INTO statement. Contains a 0 if the statement string is not a query. For DESCRIBE PROCEDURE, the number of result sets returned by the stored procedure. Contains a 0 if no result sets are returned.	The number of host variables that are used by the SQL statement.
Each SQLDA contains *stem*.SQLD of the following variables. Therefore, 1<=*n*<=*stem*.SQLD. There is one occurrence of each variable for each column of the result table or host variable that is described by the SQLDA. This set of variables is equivalent to the SQLVAR structure in the SQLDA for other languages.		
stem.n.SQLTYPE	Indicates the data type of the column or parameter and whether it can contain null values. For a description of the type codes, see "SQLTYPE and SQLLEN" on page 2090. For a distinct type, the data type on which the distinct type was based is placed in this field. The base SQLVAR provides no indication that this is part of the description of a distinct type.	Indicates the data type of the host variable and whether an indicator variable is provided. Host variables for datetime values must be character string variables. For FETCH, a datetime type code means a fixed-length character string. For a description of the type codes, see "SQLTYPE and SQLLEN" on page 2090.
stem.n.SQLLEN	For a column other than a DECIMAL or NUMERIC column, the length attribute of the column or parameter. For datetime data, the length of the string representation of the value. See "SQLTYPE and SQLLEN" on page 2090 for a description of allowable values.	For a host variable that does not have a decimal data type, the length attribute of the host variable. See "SQLTYPE and SQLLEN" on page 2090 for a description of allowable values.
stem.n.SQLLEN.SQLPRECISION	For a DECIMAL or NUMERIC column, the precision of the column or parameter.	For a host variable with a decimal data type, the precision of the host variable.
stem.n.SQLLEN.SQLSCALE	For a DECIMAL or NUMERIC column, the scale of the column or parameter.	For a host variable with a decimal data type, the scale of the host variable.

Table 172. Fields of a REXX SQLDA (continued)

Variable name	Usage in DESCRIBE and PREPARE INTO	Usage in FETCH, OPEN, EXECUTE, and CALL
stem.n.SQLCCSID	For a string column or parameter, the CCSID of the column or parameter.	For a string host variable, the CCSID of the host variable.
stem.n.SQLUSECCSID	Not used.	Set to a new CCSID. If set, REXX will change the CCSID of the SQLDATA.
stem.n.SQLLOCATOR	For DESCRIBE PROCEDURE, the value of a result set locator.	Not used.
stem.n.SQLDATA	Not used.	Before EXECUTE or OPEN, contains the value of an input host variable. The application must supply this value. After FETCH, contains the values of an output host variable.
stem.n.SQLIND	Not used.	Before EXECUTE or OPEN, contains a negative number to indicate that the input host variable in *stem.n*.SQLDATA is null. The application must supply this value. After FETCH, contains a negative number if the value of the output host variable in *stem.n*.SQLDATA is null.
stem.n.SQLNAME	The name of the *n*th column in the result table. For DESCRIBE PROCEDURE, contains the cursor name that is used by the stored procedure to return the result set. The values for SQLNAME appear in the order that the cursors were opened by the stored procedure.	Not used.

DB2 catalog tables

DB2 for z/OS maintains a set of tables (in database DSNDB06) called the DB2 catalog.

About these topics

These topics describe that catalog by describing the columns of each catalog table.

The catalog tables describe such things as table spaces, tables, columns, indexes, privileges, application plans, and packages. Authorized users can query the catalog; however, it is primarily intended for use by DB2 and is therefore subject to change. All catalog tables are qualified by SYSIBM. Do not use this qualifier for user-defined tables.

The catalog tables are updated by DB2 during normal operations in response to certain SQL statements, commands, and utilities.

Additional information

Release dependency indicators: Some objects depend on functions in particular releases of DB2. If you are running on a release of DB2 and fall back to a previous release, an object that depends on the more recent release becomes frozen. The object is marked with a release dependency indicator and is unavailable until remigration.

The release dependency indicator, which is listed in the IBMREQD, RELCREATED, and RELBOUND columns of the catalog tables, shows the release of DB2 upon which the objects depend.

Important: The IBMREQD column is not a reliable indicator for indicating release dependencies. Where possible, the RELCREATED and RELBOUND columns should be used instead.

Release dependency indicators in IBMREQD, RELCREATED, and RELBOUND are defined by the following values:

Value	Meaning
B	Version 1R3 dependency indicator, not from the machine-readable material (MRM) tape
C	Version 2R1 dependency indicator, not from MRM tape
D	Version 2R2 dependency indicator, not from MRM tape
E	Version 2R3 dependency indicator, not from MRM tape
F	Version 3R1 dependency indicator, not from MRM tape
G	Version 4 dependency indicator, not from MRM tape
H	Version 5 dependency indicator, not from MRM tape
I	Version 6 dependency indicator, not from MRM tape
J	Version 6 dependency indicator, not from MRM tape
K	Version 7 dependency indicator, not from MRM tape
L	Version 8 dependency indicator, not from MRM tape

M Version 9 dependency indicator, not from MRM tape

O Version 10 dependency indicator, not from MRM tape

N Not from MRM tape, no dependency

Programming interface information

Not all catalog table columns are part of the general-use programming interface. Whether a column is part of this interface is indicated in a column labeled "Use" in the table that describes the column. The values that "Use" can assume are as follows:

Value	Meaning
G	Column is part of the general-use programming interface
S	Column is part of the product-sensitive interface
I	Column is for internal use only
N	Column is not used

For columns for which "Use" is N or I, the name of the column and its description do not appear in the explanation of the column.

Table spaces and indexes

DB2 catalog tables are contained in certain table spaces and have indexes.

The following tables list the table space and indexes for each catalog table and lists the index fields for each index. The indexes are in ascending order, except where noted.

Table 173. Table spaces and indexes for the catalog tables

TABLE SPACE DSNDB06. ...	TABLE SYSIBM. ...	INDEX SYSIBM. ...	INDEX FIELDS
SYSALTER	SYSOBDS	DSNDB01	CREATOR.NAME.ODBTYPE
		DSNDB02	DBID.PSID
SYSTSOBX	SYSOBD_AUX	DSNOBD03	OBD_IMAGE
SYSTSATS	SYSAUTOALERTS	DSNALX01	ALERT_ID
		DSNALX02	HISTORY_ENTRY_ID
		DSNALX03	RETURN_CODE.ACTION
		DSNALX04	TARGET_QUALIFIER. TARGET_OBJECT. TARGET_PARTITION
		DSNALX05	CREATEDTS
		DSNALX06	STARTTS.RETURN_CODE
SYSTSATX	SYSAUTOALERTS_OUT	DSNALX07	AUXID.AUXVER
SYSTSATW	SYSAUTOTIMEWINDOWS	DSNTWX01	WINDOW_ID
SYSTSPRH	SYSAUTORUNS_HIST	DSNPHX01	HISTORY_ENTRY_ID
		DSNPHX02	PROC_NAME.STARTTS
		DSNPHX03	STARTTS
SYSTSPHX	SYSAUTORUNS_HISTOU	DSNPHX04	AUXID.AUXVER
SYSTSCPY	SYSCOPY	DSNUCH01	DBNAME.TSNAME.START_RBA.[1] TIMESTAMP[1]
		DSNUCX01	DSNAME
SYSCONTX	SYSCONTEXT	DSNCTX01	NAME
		DSNCTX02	SYSTEMAUTHID
		DSNCTX03	CONTEXTID
		DSNCTX04	DEFAULTROLE
	SYSCONTEXTAUTHIDS	DSNCDX01	CONTEXTID.AUTHID
		DSNCDX02	ROLE
	SYSCTXTTRUSTATTRS	DSNCAX01	CONTEXTID. NAME.VALUE
SYSTSFAU	SYSCOLAUTH	DSNACX01	CREATOR.TNAME.COLNAME
		DSNACX02	CREATOR.TNAME.TIMESTAMP
		DSNACX03	GRANTOR.GRANTORTYPE.CREATOR. TNAME.TIMESTAMP
		DSNACX04	GRANTEE.GRANTEETYPE.CREATOR. TNAME.TIMESTAMP
SYSTSCOL	SYSCOLUMNS	DSNDCX01	TBCREATOR.TBNAME.NAME

Table 173. Table spaces and indexes for the catalog tables (continued)

TABLE SPACE DSNDB06. ...	TABLE SYSIBM. ...	INDEX SYSIBM. ...	INDEX FIELDS
		DSNDCX02	TYPESCHEMA.TYPENAME
		DSNDCX05	TBCREATOR.TBNAME
SYSTSFLD	SYSFIELDS	DSNDBX02	TBCREATOR.TBNAME.NAME
SYSTSFOR	SYSFOREIGNKEYS	DSNDRH01	CREATOR.TBNAME.RELNAME
SYSTSIXC	SYSINDEXCLEANUP	DSNICX01	DBNAME.INDEXSPACE
SYSTSIXR	SYSINDEXES_RTSECT	DSNDXX06	RTSECTION
SYSTSIXS	SYSINDEXES	DSNDXX01	CREATOR.NAME
		DSNDXX02	DBNAME.INDEXSPACE
		DSNDXX03	TBCREATOR.TBNAME.CREATOR. NAME
		DSNDXX04	INDEXTYPE
		DSNDXX07	TBCREATOR.TBNAME
SYSTSIXT	SYSINDEXES_TREE	DSNDXX05	PARSETREE
SYSTSIPT	SYSINDEXPART	DSNDRX01	IXCREATOR.IXNAME.PARTITION
		DSNDRX02	STORNAME
		DSNDRX03	IXCREATOR.IXNAME
SYSTSKEY	SYSKEYS	DSNDKX01	IXCREATOR.IXNAME.COLNAME
		DSNDKX02	IXCREATOR.IXNAME
		DSNDKX03	IXCREATOR.IXNAME.COLSEQ
SYSTSREL	SYSRELS	DSNDLX01	REFTBCREATOR.REFTBNAME
		DSNDLX02	CREATOR.TBNAME
		DSNDLX03	IXOWNER.IXNAME
		DSNDLX04	CREATOR.TBNAME.RELNAME
SYSTSSYN	SYSSYNONYMS	DSNDYX01	CREATOR.NAME
		DSNDYX02	TBCREATOR.TBNAME
SYSTSTAU	SYSTABAUTH	DSNATX01	GRANTOR.GRANTORTYPE
		DSNATX02	GRANTEE.TCREATOR.TTNAME. GRANTEETYPE.UPDATECOLS. ALTERAUTH.DELETEAUTH. INDEXAUTH.INSERTAUTH. SELECTAUTH.UPDATEAUTH. CAPTUREAUTH.REFERENCESAUTH. REFCOLS.TRIGGERAUTH
		DSNATX03	GRANTEE.GRANTEETYPE.COLLID CONTOKEN
		DSNATX04	TCREATOR.TTNAME
		DSNATX05	TCREATOR.TTNAME.TIMESTAMP
SYSTSTPT	SYSTABLEPART	DSNDPX01	DBNAME.TSNAME.PARTITION
		DSNDPX02	STORNAME
		DSNDPX03	DBNAME.TSNAME.LOGICAL_PART
		DSNDPX04	IXCREATOR.IXNAME

Table 173. Table spaces and indexes for the catalog tables (continued)

TABLE SPACE DSNDB06. ...	TABLE SYSIBM. ...	INDEX SYSIBM. ...	INDEX FIELDS
		DSNDPX05	DBNAME.TSNAME
SYSTSTAB	SYSTABLES	DSNDTX01	CREATOR.NAME
		DSNDTX02	DBID.OBID.CREATOR.NAME
		DSNDTX03	TBCREATOR.TBNAME
		DSNDTX05	TBNAME.TSNAME
SYSTSTPF	SYSTABLES_PROFILES	DSNPRX01	SCHEMA.TBNAME.PROFILE_TYPE
SYSTSPTX	SYSPROFILES_TEXT	DSNPRX02	PROFILE_TEXT
SYSTSTSP	SYSTABLESPACE	DSNDSX01	DBNAME.NAME
SYSTSXTM	SYSXMLTYPMOD	DSNTMX01	XML_TYPEMOD_ID
SYSTSXTS	SYSXMLTYPMSCHEMA	DSNMSX01	XML_TYPMOD_ID. XSROBJECTID
		DSNMSX02	XSROBJECTID
SYSTSDBA	SYSDATABASE	DSNDDH01	NAME
		DSNDDX02	GROUP_MEMBER
SYSTSDBU	SYSDBAUTH	DSNADH01	GRANTEE.NAME.GRANTEETYPE
		DSNADH02	NAME
		DSNADX01	GRANTOR.NAME.GRANTORTYPE
SYSDDF	IPLIST	DSNDUX01	LINKNAME.IPADDR
	IPNAMES	DSNFPX01	LINKNAME
	LOCATIONS	DSNFCX01	LOCATION
	LULIST	DSNFLX01	LINKNAME.LUNAME
		DSNFLX02	LUNAME
	LUMODES	DSNFMX01	LUNAME.MODENAME
	LUNAMES	DSNFNX01	LUNAME
	MODESELECT	DSNFDX01	LUNAME.AUTHID[1].PLANNAME[1]
	USERNAMES	DSNFEX01	TYPE.AUTHID[1].LINKNAME[1]
SYSEBCDC	SYSDUMMY1		
	SYSDUMMYE		
SYSTSASC	SYSDUMMYA		
SYSTSUNI	SYSDUMMYU		
SYSGPAUT	SYSRESAUTH	DSNAGH01	GRANTEE.QUALIFIER. NAME.OBTYPE. GRANTEETYPE
		DSNAGX01	GRANTOR.QUALIFIER. NAME.OBTYPE. GRANTORTYPE
SYSTSSFB	SYSSTATFEEDBACK	DSNSFX01	TBCREATOR.TBNAME. IXCREATOR.IXNAME. COLNAME.COLGROUPCOLNO. NUMCOLUMNS.TYPE
		DSNSFX02	TBCREATOR.TBNAME.

Table 173. Table spaces and indexes for the catalog tables (continued)

TABLE SPACE DSNDB06. ...	TABLE SYSIBM. ...	INDEX SYSIBM. ...	INDEX FIELDS
		DSNSFX03	IXCREATOR.IXNAME.
SYSTSSTG	SYSSTOGROUP	DSNSSH01	NAME
SYSTSVOL	SYSVOLUMES	DSNSSH02	SGNAME
SYSGRTNS	SYSROUTINES_OPTS	DSNROX01	SCHEMA.ROUTINENAME. BUILDDATE.BUILDTIME
	SYSROUTINES_SRC	DSNRSX01	ROUTINENAME
		DSNRSX02	SCHEMA.ROUTINENAME. BUILDDATE. SEQNO
SYSHIST	SYSCOLDIST_HIST	DSNHFX01	TBOWNER.TBNAME. NAME.STATSTIME
	SYSCOLUMNS_HIST	DSNHEX01	TBCREATOR.TBNAME. NAME.STATSTIME
	SYSINDEXES_HIST	DSNHHX01	TBCREATOR.TBNAME. NAME.STATSTIME
		DSNHHX02	CREATOR.NAME
	SYSINDEXPART_HIST	DSNHGX01	IXCREATOR.IXNAME. PARTITION.STATSTIME
	SYSINDEXSTATS_HIST	DSNHIX01	OWNER.NAME. PARTITION.STATSTIME
	SYSLOBSTATS_HIST	DSNHJX01	DBNAME.NAME.STATSTIME
	SYSKEYTARGETS_HIST	DSNHKX01	IXSCHEMA.IXNAME. KEYSEQ.STATSTIME
	SYSKEYTGTDIST_HIST	DSNTDX02	IXSCHEMA.IXNAME KEYSEQ.STATSTIME
	SYSTABLEPART_HIST	DSNHCX01	DBNAME.TSNAME. PARTITION.STATSTIME
	SYSTABLES_HIST	DSNHDX01	CREATOR.NAME.STATSTIME
	SYSTABSTATS_HIST	DSNHBX01	OWNER.NAME. PARTITION.STATSTIME
SYSJAVA	SYSJARCONTENTS	DSNJCX01	JARSCHEMA.JAR_ID
	SYSJAROBJECTS	DSNJOX01	JARSCHEMA.JAR_ID
	SYSJAVAOPTS	DSNJVX01	JARSCHEMA.JAR_ID
	SYSJAVAPATHS	DSNJPX01	JARSCHEMA.JAR_ID.ORDINAL
		DSNJPX02	PE_JARSCHEMA.PE_JAR_ID
SYSJAUXA LOB	SYSJARDATA	DSNJDX01	JAR_DATA
SYSJAUXB LOB	SYSJARCLASS_SOURCE	DSNJSX01	CLASS_SOURCE
SYSTSAUX	SYSAUXRELS	DSNOXX01	TBOWNER.TBNAME
		DSNOXX02	AUXTBOWNER.AUXTBNAME
SYSTSCON	SYSCONSTDEP	DSNCCX01	BSCHEMA.BNAME.BTYPE
		DSNCCX02	DTBCREATOR.DTBNAME
SYSTSDAT	SYSDATATYPES	DSNODX01	SCHEMA.NAME

Table 173. Table spaces and indexes for the catalog tables (continued)

TABLE SPACE DSNDB06. ...	TABLE SYSIBM. ...	INDEX SYSIBM. ...	INDEX FIELDS
		DSNODX02	DATATYPEID[1]
SYSTSDEP	SYSDEPENDENCIES	DSNONX01	BSCHEMA.BNAME. BCOLNAME.BTYPE. DSCHEMA.DNAME. DCOLNAME.DTYPE
		DSNONX02	DSCHEMA.DNAME. DCOLNAME.DTYPE. BSCHEMA.BNAME. BCOLNAME.BTYPE
SYSTSENV	SYSENVIRONMENT	DSNOEX01	ENVID
SYSTSKYC	SYSKEYCOLUSE	DSNCUX01	TBCREATOR.TBNAME. CONSTNAME.COLSEQ.
SYSTSPRM	SYSPARMS	DSNOPX01	SCHEMA.SPECIFICNAME. ROUTINETYPE.ROWTYPE ORDINAL.VERSION
		DSNOPX02	TYPESCHEMA.TYPENAME. ROUTINETYPE.CAST_FUNCTION. OWNER.SCHEMA.SPECIFICNAME
		DSNOPX03	TYPESCHEMA.TYPENAME
		DSNOPX04	SCHEMA.SPECIFICNAME. ROUTINETYPE.VERSION
SYSTSRAU	SYSROUTINEAUTH	DSNOAX01	GRANTOR.SCHEMA. SPECIFICNAME.ROUTINETYPE. GRANTEETYPE.EXECUTEAUTH. GRANTORTYPE
		DSNOAX02	GRANTEE.SCHEMA.SPECIFICNAME. ROUTINETYPE.GRANTEETYPE. EXECUTEAUTH.GRANTEDTS
		DSNOAX03	SCHEMA.SPECIFICNAME ROUTINETYPE
SYSTSROU	SYSROUTINES	DSNOFX01	NAME.PARM_COUNT. PARM_SIGNATURE.ROUTINETYPE. SCHEMA.PARM1.PARM2.PARM3. PARM4.PARM5.PARM6.PARM7. PARM8.PARM9.PARM10.PARM11. PARM12.PARM13.PARM14.PARM15. PARM16.PARM17.PARM18.PARM19. PARM20.PARM21.PARM22.PARM23. PARM24.PARM25.PARM26.PARM27. PARM28.PARM29.PARM30. VERSION
		DSNOFX02	SCHEMA.SPECIFICNAME. ROUTINETYPE.VERSION
		DSNOFX03	NAME.SCHEMA.CAST_FUNCTION. PARM_COUNT.PARM_SIGNATURE. PARM1
		DSNOFX04	ROUTINEID[1]

Table 173. Table spaces and indexes for the catalog tables (continued)

TABLE SPACE DSNDB06. ...	TABLE SYSIBM. ...	INDEX SYSIBM. ...	INDEX FIELDS
		DSNOFX05	SOURCESCHEMA.SOURCESPECIFIC. ROUTINETYPE
		DSNOFX06	SCHEMA.NAME.ROUTINETYPE. PARM_COUNT
		DSNOFX07	NAME.PARM_COUNT. ROUTINETYPE.SCHEMA. PARM_SIGNATURE. PARM1.PARM2.PARM3. PARM4.PARM5.PARM6.PARM7. PARM8.PARM9.PARM10.PARM11. PARM12.PARM13.PARM14.PARM15. PARM16.PARM17.PARM18.PARM19. PARM20.PARM21.PARM22.PARM23. PARM24.PARM25.PARM26.PARM27. PARM28.PARM29.PARM30. VERSION
		DSNOFX08	JARSCHEMA. JAR_ID
SYSTSSCM	SYSSCHEMAAUTH	DSNSKX01	GRANTEE.SCHEMANAME. GRANTEETYPE
		DSNSKX02	GRANTOR.GRANTORTYPE
SYSTSTBC	SYSTABCONST	DSNCNX01	TBCREATOR.TBNAME.CONSTNAME
		DSNCNX02	IXOWNER IXNAME
SYSTSTRG	SYSTRIGGERS	DSNOTX01	SCHEMA.NAME.SEQNO
		DSNOTX02	TBOWNER.TBNAME
		DSNOTX03	SCHEMA.TRIGNAME
SYSTSPKG	SYSPACKAGE	DSNKKX01	LOCATION.COLLID.NAME. VERSION
		DSNKKX02	LOCATION.COLLID.NAME. CONTOKEN
SYSTSPKC	SYSPACKCOPY	DSNPCX01	LOCATION.COLLID. NAME.CONTOKEN. COPYID
SYSTSPKA	SYSPACKAUTH	DSNKAX01	GRANTOR.LOCATION.COLLID.NAME. GRANTORTYPE
		DSNKAX02	GRANTEE.LOCATION.COLLID. NAME.BINDAUTH.COPYAUTH. EXECUTEAUTH.GRANTEETYPE
		DSNKAX03	LOCATION.COLLID.NAME
SYSTSPKD	SYSPACKDEP	DSNKDX01	DLOCATION.DCOLLID.DNAME. DCONTOKEN
		DSNKDX02	BQUALIFIER.BNAME.BTYPE
		DSNKDX03	BQUALIFIER.BNAME.BTYPE. DTYPE
SYSTSPKL	SYSPACKLIST	DSNKLX01	LOCATION.COLLID.NAME

Table 173. Table spaces and indexes for the catalog tables (continued)

TABLE SPACE DSNDB06. ...	TABLE SYSIBM. ...	INDEX SYSIBM. ...	INDEX FIELDS
		DSNKLX02	PLANNAME.SEQNO.LOCATION. COLLID.NAME
SYSTSPKS	SYSPACKSTMT	DSNKSX01	LOCATION.COLLID.NAME. CONTOKEN.STMTNOI.SECTNOI. SEQNO
SYSTSPKY	SYSPKSYSTEM	DSNKYX01	LOCATION.COLLID.NAME. CONTOKEN.SYSTEM.ENABLE
SYSTSPLY	SYSPLSYSTEM	DSNKPX01	NAME.SYSTEM.ENABLE
SYSTSDBR	SYSDBRM	DSNDBX01	PLNAME
		DSNDFX01	PLNAME.NAME
SYSTSPLN	SYSPLAN	DSNPPH01	NAME
SYSTSPLA	SYSPLANAUTH	DSNAPH01	GRANTEE.NAME.EXECUTEAUTH. GRANTEETYPE
		DSNAPX01	GRANTOR.GRANTORTYPE
		DSNAPX02	NAME
SYSTSPLD	SYSPLANDEP	DSNGGX01	BCREATOR.BNAME.BTYPE
		DSNGGX05	DNAME
SYSTSQRA	SYSQUERY_AUX	DSNQSX01	STMTTEXT
SYSTSQRY	SYSQUERY	DSNQYX01	QUERY_HASH. SCHEMA.SOURCE. QUERY_SEC_HASH
		DSNQYX02	QUERYID
		DSNQYX03	LOCATION. COLLECTION. PACKAGE. VERSION.SECTNO
SYSTSQRP	SYSQUERYPLAN	DSNQNX01	QUERYID.COPYID
SYSTSQRO	SYSQUERYOPTS	DSNQPX01	QUERYID.COPYID
SYSTSSTM	SYSSTMT	DSNPSX01	PLNAME.NAME
		DSNPSX02	PLNAME.NAME.SEQNO
SYSPLUXA	SYSROUTINESTEXT	DSNPLX01	TEXT
SYSPLUXB	SYSROUTINES_TREE	DSNPLX02	PTREE
SYSROLES	SYSOBJROLEDEP	DSNRDX01	DSCHEMA.DNAME.DTYPE
		DSNRDX02	ROLENAME
	SYSROLES	DSNRLX01	NAME
SYSTSTSS	SYSTABLESPACESTATS	DSNRTX01	DBID.PSID.PARTITION.INSTANCE
SYSTSISS	SYSINDEXSPACESTATS	DSNRTX02	DBID.ISOBID.PARTITION.INSTANCE
		DSNRTX03	CREATOR.NAME
SYSSEQ	SYSSEQUENCES	DSNSQX01	SCHEMA.NAME
		DSNSQX02	SEQUENCEID[1]
		DSNSQX03	SEQSCHEMA.SEQNAME

Table 173. Table spaces and indexes for the catalog tables (continued)

TABLE SPACE DSNDB06. ...	TABLE SYSIBM. ...	INDEX SYSIBM. ...	INDEX FIELDS
SYSSEQ2	SYSSEQUENCEAUTH	DSNWCX01	SCHEMA.NAME
		DSNWCX02	GRANTOR.SCHEMA.NAME. GRANTORTYPE
		DSNWCX03	GRANTEE.SCHEMA.NAME. GRANTEETYPE
	SYSSEQUENCESDEP	DSNSRX01	DCREATOR.DNAME.DCOLNAME
		DSNSRX02	BSCHEMA.BNAME.DTYPE
SYSSTATS	SYSCOLDIST	DSNTNX01	TBOWNER.TBNAME.NAME
	SYSCOLDISTSTATS	DSNTPX01	TBOWNER.TBNAME.NAME PARTITION
	SYSCOLSTATS	DSNTCX01	TBOWNER.TBNAME.NAME PARTITION
	SYSINDEXSTATS	DSNTXX01	OWNER.NAME.PARTITION
	SYSKEYTARGETSTATS	DSNTKX01	IXSCHEMA.IXNAME.KEYSEQ. PARTITION
	SYSKEYTGTDIST	DSNTDX01	IXSCHEMA.IXNAME.KEYSEQ
	SYSKEYTGTDISTSTATS	DSNTSX01	IXSCHEMA.IXNAME.KEYSEQ. PARTITION
	SYSLOBSTATS	DSNLNX01	DBNAME.NAME
	SYSTABSTATS	DSNTTX01	OWNER.NAME.PARTITION
		DSNTTX02	(DBNAME, TSNAME, PARTITION)
SYSTSSRG	SYSSTRINGS	DSNSSX01	OUTCCSID.INCCSID.IBMREQD
SYSTSCKS	SYSCHECKS	DSNSCX01	TBOWNER.TBNAME.CHECKNAME
SYSTSCHX	SYSCHECKS2	DSNCHX01	TBOWNER.TBNAME.CHECKNAME
SYSTSCKD	SYSCHECKDEP	DSNSDX01	TBOWNER.TBNAME.CHECKNAME COLNAME
SYSTARG	SYSKEYTARGETS	DSNRKX01	IXSCHEMA.IXNAME. KEYSEQ
		DSNRKX02	DATATYPEID. XMLPATTERN_INTERNAL
SYSTSADT	SYSAUDITPOLICIES	DSNAPX03	AUDITPOLICYNAME
SYSTSCTL	SYSCONTROLS	DSNCLX01	SCHEMA.NAME
		DSNCLX02	CONTROL_ID
		DSNCLX03	TBSCHEMA.TBNAME
		DSNCLX04	TBSCHEMA.TBNAME. ENABLE
		DSNCLX05	TBSCHEMA.TBNAME. ENABLE. CONTROL_TYPE
SYSTSCTD	SYSCONTROLS_DESC	DSNTRX02	DESCRIPTOR
SYSTSCTR	SYSCONTROLS_RTXT	DSNTRX01	RULETEXT

Table 173. Table spaces and indexes for the catalog tables (continued)

TABLE SPACE DSNDB06. ...	TABLE SYSIBM. ...	INDEX SYSIBM. ...	INDEX FIELDS
SYSTSPEN	SYSPENDINGDDL	DSNPDX01	DBNAME.TSNAME. CREATEDTS. OPTION_SEQNO
		DSNPDX02	OBJSCHEMA.OBJNAME. OBJTYPE.TSNAME. CREATEDTS. OPTION_SEQNO
SYSTSPDT	SYSPENDINGDDLTEXT	DSNPDX03	STATEMENT_TEXT
SYSTSPDO	SYSPENDINGOBJECTS	DSNPOX01	DBNAME.TSNAME. PARTITION.COLNAME
		DSNPOX02	OBJSCHEMA.OBJNAME. OBJTYPE
		DSNPOX03	DNAME.INDEXSPACE
SYSTSPKX	SYSPACKSTMT_STMT	DSNPKX01	STATEMENT
SYSTSPVR	SYSPACKSTMT_STMB	DSNKSX02	STMTBLOB
SYSTSTRT	SYSTRIGGERS_STMT	DSNOTX04	STATEMENT
SYSTSVAD	SYSVARIABLES_DESC	DSNOVX03	DESCRIPTOR
SYSTSVAR	SYSVARIABLES	DSNOVX01	SCHEMA.NAME
SYSTSVAT	SYSVARIABLES_TEXT	DSNOVX02	DEFAULTTEXT
SYSTSVAU	SYSVARIABLEAUTH	DSNVAX01	GRANTEE.GRANTEETYPE. SCHEMA.NAME
		DSNVAX02	GRANTOR.GRANTORTYPE. SCHEMA.NAME
		DSNVAX03	SCHEMA.NAME
SYSTSVTR	SYSVIEWS_TREE	DSNVWX02	PARSETREE
SYSTSVWT	SYSVIEWS_STMT	DSNVWX01	STATEMENT
SYSUSER	SYSUSERAUTH	DSNAUH01	GRANTEE GRANTEDTS. GRANTEETYPE
		DSNAUX02	GRANTOR.GRANTORTYPE
SYSTSVWD	SYSVIEWDEP	DSNGGX02	BCREATOR.BNAME.BTYPE
		DSNGGX03	BSCHEMA.BNAME.BTYPE
		DSNGGX04	BCREATOR.BNAME.BTYPE.DTYPE
		DSNGGX06	DCREATOR.DNAME.DTYPE
SYSTSVEW	SYSVIEWS	DSNVVX01	CREATOR.NAME.SEQNO.TYPE
SYSXML	SYSXMLRELS	DSNXRX01	TBOWNER.TBNAME.COLNAME
		DSNXRX02	XMLTBOWNER.XMLTBNAME
	SYSXMLSTRINGS	DSNXSX01	STRINGID
		DSNXSX02	STRING

Note: 1. Index field is in descending order

SQL statements allowed on the catalog

Certain SQL statements can be used to change the value of certain options for existing catalog indexes, sequences, and table spaces, or to add indexes to any of the catalog tables.

Table 174. SQL statements that can be used to change existing catalog indexes, sequences, and table spaces, or to add indexes to any of the catalog tables

SQL statement	Index	Allowable clauses and usage notes
ALTER INDEX	IBM-defined	Only these clauses are allowed: CLOSE COPY FREEPAGE GBPCACHE NOT PADDED PADDED PCTFREE PIECESIZE You cannot alter the GBPCACHE value for indexes DSNDXX01, DSNDXX02, and DSNDXX03, which are on catalog table SYSIBM.SYSINDEXES.
ALTER INDEX	User-created	All clauses are allowed, except for the following: BUFFERPOOL REGENERATE COMPRESS YES Any partitioning clause
ALTER SEQUENCE		The only clause allowed is MAXVALUE. You can only change the MAXVALUE value of the catalog sequence DSNSEQ_IMPLICITDB. The only value specific must be an integer between 1 and 60000, inclusive.
ALTER TABLE		The only clause allowed is DATA CAPTURE CHANGES.

Table 174. SQL statements that can be used to change existing catalog indexes, sequences, and table spaces, or to add indexes to any of the catalog tables (continued)

SQL statement	Index	Allowable clauses and usage notes
ALTER TABLESPACE		Only these clauses are allowed: CLOSE FREEPAGE GBPCACHE LOCKMAX MAXROWS PCTFREE TRACKMOD You cannot alter the GBPCACHE or MAXROWS value of some catalog table spaces. Do not specify GBPCACHE for the following table spaces: • DSNDB06.SYSTSCOL • DSNDB06.SYSTSDBA • DSNDB06.SYSTSDBR • DSNDB06.SYSTSDBU • DSNDB06.SYSTSFAU • DSNDB06.SYSTSFLD • DSNDB06.SYSTSFOR • DSNDB06.SYSTSIPT • DSNDB06.SYSTSIXR • DSNDB06.SYSTSIXS • DSNDB06.SYSTSIXT • DSNDB06.SYSTSKEY • DSNDB06.SYSTSPKA • DSNDB06.SYSTSPKD • DSNDB06.SYSTSPKG • DSNDB06.SYSTSPKL • DSNDB06.SYSTSPKS • DSNDB06.SYSTSPKX • DSNDB06.SYSTSPKY • DSNDB06.SYSTSPLA • DSNDB06.SYSTSPLD • DSNDB06.SYSTSPLN • DSNDB06.SYSTSPLY • DSNDB06.SYSTSPVR • DSNDB06.SYSTSREL • DSNDB06.SYSTSSTM • DSNDB06.SYSTSSYN • DSNDB06.SYSTSTAB • DSNDB06.SYSTSTAU • DSNDB06.SYSTSTPT • DSNDB06.SYSTSTSP For DSNDB06.SYSSEQ, MAXROW can be specified only with a value of 1. You can specify the LOCKSIZE keyword on the ALTER TABLESPACE statement for any catalog table spaces that are not LOB table spaces.

Table 174. SQL statements that can be used to change existing catalog indexes, sequences, and table spaces, or to add indexes to any of the catalog tables (continued)

SQL statement	Index	Allowable clauses and usage notes
CREATE INDEX	User-created	All clauses are allowed, except for: CLOSE YES CLUSTER UNIQUE DEFER YES (only on tables SYSINDEXES, SYSINDEXPART, and SYSKEYS) COMPRESS YES Any partitioning clause Indexes that are created with *key-expressions* are not allowed on the catalog. The only value allowed for BUFFERPOOL is BP0. You can create up to 500 indexes on the catalog.
DROP INDEX	User-created	The statement has no clauses.

Reorganizing the catalog

The REORG TABLESPACE utility can be run on all the table spaces in the catalog database (DSNDB06) to reclaim unused or wasted space, which can affect performance.

The utility observes the PCTFREE and FREEPAGE values specified in the ALTER INDEX statement for all the catalog indexes and the following table spaces:

- DSNDB06.SYSDDF
- DSNDB06.SYSGPAUT
- DSNDB06.SYSGRTNS
- DSNDB06.SYSHIST
- DSNDB06.SYSJAVA
- DSNDB06.SYSJAUXA
- DSNDB06.SYSJAUXB
- DSNDB06.SYSSEQ
- DSNDB06.SYSSEQ2
- DSNDB06.SYSSTATS
- DSNDB06.SYSTSCHX
- DSNDB06.SYSTSCKD
- DSNDB06.SYSTSCKS
- DSNDB06.SYSTSCPY
- DSNDB06.SYSTSSRG
- DSNDB06.SYSTSUSR
- DSNDB01.SCT02
- DSNDB01.SPT01

For details on running REORG TABLESPACE, see *DB2 Utility Guide and Reference*.

New and changed catalog tables

This release of DB2 for z/OS includes many new catalog tables as well as many catalog tables that have new or changed content.

Descriptions of the following catalog tables have been added:

Table 175. New catalog tables

Catalog table name	Description
"SYSIBM.SYSINDEXCLEANUP table" on page 2210	The rows in the SYSIBM.SYSINDEXCLEANUP table specify time windows to control index cleanup processing. Each row specifies a time window to enable or disable the cleanup of pseudo-deleted index entries for specific database objects.
"SYSIBM.SYSQUERYPREDICATE table" on page 2332	The SYSIBM.SYSQUERYPREDICATE table contains information about predicates for queries in the SYSIBM.SYSQUERY table that have been identified for extended optimization. It correlates to the SYSIBM.SYSQUERY table by the QUERYID column.
"SYSIBM.SYSQUERYSEL table" on page 2337	The SYSIBM.SYSQUERYSEL table contains information about the selectivity of predicates for queries in the SYSIBM.SYSQUERY table that have been identified for extended optimization. It correlates to the SYSIBM.SYSQUERY table by the QUERYID column.
"SYSIBM.SYSSTATFEEDBACK table" on page 2370	The SYSIBM.SYSTATFEEDBACK table contains information about missing or conflicting catalog statistics for SQL statements.
"SYSIBM.SYSVARIABLEAUTH table" on page 2432	The SYSIBM.SYSVARIABLEAUTH table contains one row for each privilege of each authorization ID that has privileges on a global variable.
"SYSIBM.SYSVARIABLES_DESC table" on page 2434	The SYSIBM.SYSVARIABLES_DESC table is an auxiliary table for the SYSIBM.SYSVARIABLES table.
"SYSIBM.SYSVARIABLES table" on page 2429	The SYSIBM.SYSVARIABLES table contains one row for each global variable that is created.
"SYSIBM.SYSVARIABLES_TEXT table" on page 2435	The SYSIBM.SYSVARIABLES_TEXT table is an auxiliary table for the DEFAULTTEXT column of the SYSIBM.SYSVARIABLES table.

The catalog tables that are listed in the following table have new or revised columns, column values, or column descriptions to support the new function in this release of DB2 for z/OS.

Table 176. Summary of new and revised catalog table columns

Catalog table name	New column	Revised column
"SYSIBM.SYSCHECKS table" on page 2143		RBA
"SYSIBM.SYSCOPY table" on page 2176	MODECREATED	• PIT_RBA • START_RBA • ICTYPE • STYPE • TTYPE
"SYSIBM.SYSDATATYPES table" on page 2191	• ARRAYLENGTH • ARRAYINDEXTYPEID • ARRAYINDEXTYPELEN • ARRAYINDEXSUBTYPE	• LENGTH • METATYPE • SCALE • SUBTYPE

Table 176. Summary of new and revised catalog table columns (continued)

Catalog table name	New column	Revised column
"SYSIBM.SYSDEPENDENCIES table" on page 2198	DVERSION	• BTYPE • DTYPE
"SYSIBM.SYSINDEXES table" on page 2211		• CLUSTERED • CLUSTERRATIO • COPYLRSN • SPARSE
"SYSIBM.SYSINDEXES_HIST table" on page 2217		• AVGKEYLEN • CLUSTERRATIOF • DATAREPEATFACTORF
"SYSIBM.SYSINDEXPART table" on page 2221	RBA_FORMAT	• AVGKEYLEN • CARDF • FAROFFPOSF • NEAROFFPOSF
"SYSIBM.SYSINDEXPART_HIST table" on page 2226		• AVGKEYLEN • CARDF • FAROFFPOSF • NEAROFFPOSF
"SYSIBM.SYSINDEXSPACESTATS table" on page 2229		COPYLRSN
"SYSIBM.SYSINDEXSTATS table" on page 2235		• CLUSTERRATIO • CLUSTERRATIOF • DATAREPEATFACTORF • KEYCOUNT • KEYCOUNTF
"SYSIBM.SYSINDEXSTATS_HIST table" on page 2237		• CLUSTERRATIOF • DATAREPEATFACTORF • KEYCOUNTF
"SYSIBM.SYSKEYTARGETS table" on page 2247	DESCRIPTOR	
"SYSIBM.SYSPACKAGE table" on page 2265	• APPLCOMPAT • ARCHIVESENSITIVE • BUSTIMESENSITIVE • DESCSTAT • EXTSEQNO	• APREUSE • DBPROTOCOL • SYSTIMESENSITIVE
"SYSIBM.SYSPACKCOPY table" on page 2275	• APPLCOMPAT • ARCHIVESENSITIVE • BUSTIMESENSITIVE • DESCSTAT • EXTSEQNO • SYSTIMESENSITIVE	• DBPROTOCOL • RECORDTEMPORALHIST • SYSTIMESENSITIVE
"SYSIBM.SYSPACKSTMT table" on page 2290	EXPANSION_REASON	

Table 176. Summary of new and revised catalog table columns (continued)

Catalog table name	New column	Revised column
"SYSIBM.SYSPARMS table" on page 2297		• CCSID • ENCODING_SCHEME • LENGTH • SUBTYPE
"SYSIBM.SYSPENDINGDDL table" on page 2301	• COLNAME • COLUMN_KEYWORD • PARTITION • PARTITION_KEYWORD	• CREATEDTS • OBJTYPE • OPTION_KEYWORD • OPTION_VALUE • STATEMENT_TYPE
"SYSIBM.SYSPLAN table" on page 2306	PROGAUTH	
"SYSIBM.SYSQUERY table" on page 2315	• ACCESS_PATH_HINT • OPTION_OVERRIDE • SELECTIVITY_VALID • SELECTVTY_OVERRIDE	
"SYSIBM.SYSQUERYPLAN table" on page 2321	EXPANSION_REASON	
"SYSIBM.SYSRESAUTH table" on page 2341		QUALIFIER
"SYSIBM.SYSROUTINES table" on page 2346		COMMIT_ON_RETURN
"SYSIBM.SYSSEQUENCES table" on page 2366	• SEQNAME • SEQSCHEMA	
"SYSIBM.SYSTABLEPART table" on page 2387	• PCTFREE_UDP • PCTRFREE_UPD_CALC • RBA_FORMAT	
"SYSIBM.SYSTABLES table" on page 2396	• ARCHIVING_SCHEMA • ARCHIVING_TABLE • STATS_FEEDBACK	• RBA1 • RBA2 • VERSION
"SYSIBM.SYSTABLESPACESTATS table" on page 2410	• UPDATESIZE • LASTDATACHANGE	COPYUPDATELRSN

SYSIBM.IPLIST table

The SYSIBM.IPLIST table allows multiple IP addresses to be specified for a given LOCATION.

Insert rows into this table when you want to define a remote DB2 data sharing group. The same value for the IPADDR column cannot appear in both the SYSIBM IPNAMES table and the SYSIBM.IPLIST table. Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
LINKNAME	VARCHAR(24) NOT NULL	This column is associated with the value specified in the LINKNAME column in the SYSIBM.LOCATIONS table and the SYSIBM.IPNAMES table. The values of the other columns in the SYSIBM.IPNAMES table apply to the server identified by the LINKNAME column in this row.	G
IPADDR	VARCHAR(254) NOT NULL	This column contains an IPv4 or IPv6 address, or domain name of a remote TCP/IP host of the server. If WLM Domain Name Server workload balancing is used, this column must contain the member specific domain name. If Dynamic VIPA workload balancing is used, this column must contain the member specific Dynamic VIPA address. The IPADDR column must be specified as follows: • An IPv4 address must be left justified and is represented as a dotted decimal address. For example, '123.456.78.912' would be interpreted as an IPv4 address. • An IPv6 address must be left justified and is represented as a colon hexadecimal address. An example of an IPv6 address is '2001:0DB8:0000:0000:0008:0800:200C:417A', which can also be expressed in compressed form as '2001:DB8::8:800:200C:417A'. • A domain name is converted to an IP address by the domain name server where a resulting IPv4 or IPv6 address is determined. An example of a domain name is 'stlmvs1.svl.ibm.com'.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

SYSIBM.IPNAMES table

The SYSIBM.IPNAMES table defines the remote DRDA servers DB2 can access using TCP/IP.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
LINKNAME	VARCHAR(24) NOT NULL	The value specified in this column must match the value specified in the LINKNAME column of the associated row in SYSIBM.LOCATIONS.	G
SECURITY_OUT	CHAR(1) NOT NULL WITH DEFAULT 'A'	This column defines the DRDA security option that is used when local DB2 SQL applications connect to any remote server associated with this TCP/IP host: **A** The option is “already verified”. Outbound connection requests contain an authorization ID and no password. The authorization ID used for an outbound request is either the DB2 user's authorization ID or a translated ID, depending upon the value of the USERNAMES column. The authorization ID is not encrypted when it is sent to the partner. For encryption, refer to 'D'. **D** The option is “userid and security-sensitive data encryption”. Outbound connection requests contain an authorization ID and no password. The authorization ID used for an outbound request is either the DB2 user's authorization ID or a translated ID, depending upon the value of the USERNAMES column. This option indicates that the userid and security-sensitive data are to be encrypted. For non-encryption, refer to 'A'. **E** The option is “userid, password, and security-sensitive data encryption”. Outbound connection requests contain an authorization ID and a password. The password is obtained from the SYSIBM.USERNAMES table. The USERNAMES column must specify 'O'. This option indicates that the userid, password, and security-sensitive data are to be encrypted. For non-security-sensitive data encryption, refer to 'P'. If the applications connect to any remote server as trusted, the USERNAMES column must specify 'O' or 'S'.	G

Column name	Data type	Description		Use
SECURITY_OUT (continued)		**P**	The option is “password”. Outbound connection requests contain an authorization ID and a password. The password is obtained from the SYSIBM.USERNAMES table. The USERNAMES column must specify 'O'. This option indicates that the userid and the password are to be encrypted if cryptographic services are available at the requester and if the server supports encryption. Otherwise, the userid and the password are sent to the partner in clear text. For security-sensitive data encryption, see 'E'. If the applications connect to any remote server as trusted, the USERNAMES column must specify 'O' or 'S'.	
		R	The option is “RACF PassTicket”. Outbound connection requests contain a userid and a RACF PassTicket. The value specified in the LINKNAME column is used as the RACF PassTicket application name for the remote server. The authorization ID used for an outbound request is either the DB2 user's authorization ID or a translated ID, depending upon the value of the USERNAMES column. The authorization ID is not encrypted when it is sent to the partner.	
USERNAMES	CHAR(1) NOT NULL WITH DEFAULT	This column controls outbound authorization ID translation. Outbound translation is performed when an authorization ID is sent by DB2 to a remote server.		G
		O	An outbound ID is subject to translation. Rows in the SYSIBM.USERNAMES table are used to perform ID translation. No translation or “come from” checking is performed on inbound IDs.	
		S	Row in the SYSIBM.USERNAMES table is used to obtain the system AUTHID used to establish a trusted connection.	
		blank	No translation occurs.	
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.		G

Column name	Data type	Description	Use
IPADDR	VARCHAR(254) NOT NULL WITH DEFAULT	This column contains an IPv4 or IPv6 address, or domain name of a remote TCP/IP host. The IPADDR column must be specified as follows: • An IPv4 address must be left justified and is represented as a dotted decimal address. For example, '123.456.78.91' would be interpreted as an IPv4 address. • An IPv6 address must be left justified and is represented as a colon hexadecimal address. An example of an IPv6 address is '2001:0DB8:0000:0000:0008:0800:200C:417A', which can also be expressed in compressed form as '2001:DB8::8:800:200C:417A'. • A domain name is converted to an IP address by the domain name server where a resulting IPv4 or IPv6 address is determined. An example of a domain name is 'stlmvs1.svl.ibm.com'.	G

SYSIBM.LOCATIONS table

The SYSIBM.LOCATIONS table contains a row for every accessible remote server. The row associates a LOCATION name with the TCP/IP or SNA network attributes for the remote server. Requesters are not defined in this table.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
LOCATION	VARCHAR(128) NOT NULL	A unique location name for the accessible server. This is the name by which the remote server is known to local DB2 SQL applications.	G
LINKNAME	VARCHAR(24) NOT NULL	Identifies the VTAM® or TCP/IP attributes associated with this location. For any LINKNAME specified, one or both of the following statements must be true: • A row exists in SYSIBM.LUNAMES whose LUNAME matches the value specified in the SYSIBM.LOCATIONS LINKNAME column. This row specifies the VTAM communication attributes for the remote location. • A row exists in SYSIBM.IPNAMES whose LINKNAME matches the value specified in the SYSIBM.LOCATIONS LINKNAME column. This row specifies the TCP/IP communication attributes for the remote location.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
PORT	VARCHAR(96) NOT NULL WITH DEFAULT	TCP/IP is used for outbound DRDA connections when the following statement is true: • A row exists in SYSIBM.IPNAMES, where the LINKNAME column matches the value specified in the SYSIBM.LOCATIONS LINKNAME column. If the above mentioned row is found, the value of the PORT column is interpreted as follows: • If PORT is blank, the default DRDA port (446) is used. • If PORT is nonblank, the value specified for PORT can take one of two forms: – If the value in PORT is left justified with 1-5 numeric characters, the value is assumed to be the TCP/IP port number of the remote database server. – Any other value is assumed to be a TCP/IP service name, which can be converted to a TCP/IP port number using the TCP/IP `getservbyname` socket call. TCP/IP service names are not case sensitive.	G

Column name	Data type	Description	Use
TPN	VARCHAR(192) NOT NULL WITH DEFAULT	Used only when the local DB2 begins an SNA conversation with another server. When used, TPN indicates the SNA LU 6.2 transaction program name (TPN) that will allocate the conversation. A length of zero for the column indicates the default TPN. For DRDA conversations, this is the DRDA default, which is X'07F6C4C2'. For DB2 private protocol conversations, this column is not used. When the server is DB2 Server for VSE & VM, TPN should contain the resource ID of that machine.	G
DBALIAS	VARCHAR(128) NOT NULL	Database alias. The name associated with the server. This name is used to access a remote database server. If DBALIAS is blank, the location name is used to access the remote database server. This column does not change the name of any database objects sent to the remote site that contains the location qualifier. This column applies only to DRDA connections.	G
TRUSTED	CHAR(1) NOT NULL WITH DEFAULT 'N'	Indicates whether the connection to the remote server can be trusted. This is restricted to TCP/IP only. This column is ignored for connections using SNA. **Y** Location is trusted. Access to the remote location requires trusted context defined at the remote location. **N** Location is not trusted.	G
SECURE	CHAR(1) NOT NULL WITH DEFAULT 'N'	Indicates the use of the Secure Socket Layer (SSL) protocol for outbound DRDA connections when local DB2 applications connects to the remote database server using TCP/IP. **Y** Indicates a secure connection using SSL is required for the outbound DRDA connection. **N** Indicates a secure connection is not required for the outbound DRDA connection.	G

SYSIBM.LULIST table

The SYSIBM.LULIST table allows multiple LU names to be specified for a given LOCATION.

Insert rows into this table when you want to define a remote DB2 data sharing group. The same value for LUNAME column cannot appear in both the SYSIBM.LUNAMES table and the SYSIBM.LULIST table. Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
LINKNAME	VARCHAR(24) NOT NULL	The value of the LINKNAME column in the SYSIBM.LOCATIONS table with which this row is associated. This is also the value of the LUNAME column in the SYSIBM.LUNAMES table. The values of the other columns in the SYSIBM.LUNAMES row apply to the LU identified by the LUNAME column in this row of SYSIBM.LULIST.	G
LUNAME	VARCHAR(24) NOT NULL	The VTAM logical unit name (LUNAME) of the remote database system. This LUNAME must not exist in the LUNAME column of SYSIBM.LUNAMES.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

SYSIBM.LUMODES table

Each row of the SYSIBM.LUMODES table provides VTAM with conversation limits for a specific combination of LUNAME and MODENAME. The table is accessed only during the initial conversation limit negotiation between DB2 and a remote LU. This negotiation is called *change-number-of-sessions* (CNOS) processing.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
LUNAME	VARCHAR(24) NOT NULL	LU name of the server involved in the CNOS processing.	G
MODENAME	VARCHAR(24) NOT NULL	Name of a logon mode description in the VTAM logon mode table.	G
CONVLIMIT	SMALLINT NOT NULL	Maximum number of active conversations between the local DB2 and the other system for this mode. Used to override the number in the DSESLIM parameter of the VTAM APPL definition statement for this mode.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

SYSIBM.LUNAMES table

The SYSIBM.LUNAMES table must contain a row for each remote SNA client or server that communicates with DB2.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
LUNAME	VARCHAR(24) NOT NULL	Name of the LU for one or more accessible systems. A blank string indicates the row applies to clients whose LU name is not specifically defined in this table. All other column values for a given row in this table are for clients and servers associated with the row's LU name.	G
SYSMODENAME	VARCHAR(24) NOT NULL WITH DEFAULT	Mode used to establish inter-system conversations. A blank indicates the default mode IBMDB2LM (DB2 private protocol access and for collecting sysplex balancing information from remote data sharing groups). If private protocols are used to access a remote DB2 LU or if the remote LU is a member of a DB2 data sharing group, use a separate mode other than the default mode.	G
SECURITY_IN	CHAR(1) NOT NULL WITH DEFAULT 'A'	This column defines the security options that are accepted by this DB2 when an SNA client connects to DB2: **V** The option is “verify”. An incoming connection request must include one of the following: a userid and password, a userid and RACF PassTicket, or a Kerberos security ticket. **A** The option is “already verified”. A request does not need a password, although a password is checked if it is sent. With this option, an incoming connection request is accepted if it includes any of the following: a userid, a userid and password, a userid and RACF PassTicket, or a Kerberos security ticket. If the USERNAMES column contains 'I' or 'B', RACF is not invoked to validate incoming connection requests that contain only a userid unless one of the following situations is true: • The RACF access control authorization exit (DSNX@XAC) is enabled • The IBM supplied RACF SECLABEL resource class is active.	G

Column name	Data type	Description	Use
SECURITY_OUT	CHAR(1) NOT NULL WITH DEFAULT 'A'	This column defines the security option that is used when local DB2 SQL applications connect to any remote server associated with this LUNAME: **A** The option is “already verified”. Outbound connection requests contain an authorization ID and no password. The authorization ID used for an outbound request is either the DB2 user's authorization ID or a translated ID, depending upon the value of the USERNAMES column. **R** The option is “RACF PassTicket”. Outbound connection requests contain a userid and a RACF PassTicket. The server's LU name is used as the RACF PassTicket application name. The authorization ID used for an outbound request is either the DB2 user's authorization ID or a translated ID, depending upon the value of the USERNAMES column. **P** The option is “password”. Outbound connection requests contain an authorization ID and a password. The password is obtained from the SYSIBM.USERNAMES table or RACF, depending upon the value specified in the ENCRYPTPSWDS column. The USERNAMES column must specify 'B' or 'O'.	G
ENCRYPTPSWDS	CHAR(1) NOT NULL WITH DEFAULT 'N'	This column only applies to DB2 for z/OS partners. It is provided to support connectivity to prior releases of DB2 that are unable to support RACF PassTickets. For connections between DB2 Version 5 and later, use the SECURITY_OUT='R' option instead of the ENCRYPTPSWDS='Y' option. **N** No, passwords are not in internal RACF encrypted format. This is the default. **Y** Yes for outbound requests, the encrypted password is extracted from RACF and sent to the server. For inbound requests, the password is treated as encrypted.	G
MODESELECT	CHAR(1) NOT NULL WITH DEFAULT 'N'	Whether to use the SYBIBM.MODESELECT table: **N** Use default modes: IBMDB2LM (for DB2 private protocol) and IBMRDB (for DRDA). **Y** Searches SYSIBM.MODESELECT for appropriate mode name.	G

Column name	Data type	Description	Use
USERNAMES	CHAR(1) NOT NULL WITH DEFAULT	This column controls inbound and outbound authorization ID translation, and "come from" checking. Inbound translation and "come from" checking are performed when an authorization ID is received from a remote client. Outbound translation is performed when an authorization ID is sent by DB2 to a remote server. When 'I', 'O', or 'B' is specified in this column, rows in the SYSIBM.USERNAMES table are used to perform ID translation. **I** An inbound ID is subject to translation and "come from" checking. No translation is performed on outbound IDs. **O** No translation or "come from" checking is performed on inbound IDs. An outbound ID is subject to translation. **B** An inbound ID is subject to translation and "come from" checking. An outbound ID is subject to translation. **blank** No translation occurs.	G
GENERIC	CHAR(1) NOT NULL WITH DEFAULT 'N'	Indicates whether DB2 should use its real LU name or generic LU name to identify itself to the partner LU, which is identified by this row. **N** The real VTAM LU name of this DB2 subsystem **Y** The VTAM generic LU name of this DB2 subsystem	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

SYSIBM.MODESELECT table

The SYSIBM.MODESELECT table associates a mode name with any conversation created to support an outgoing SQL request. Each row represents one or more combinations of LUNAME, authorization ID, and application plan name.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
AUTHID	VARCHAR(128) NOT NULL WITH DEFAULT	Authorization ID of the SQL request. Blank (the default) indicates that the MODENAME specified for the row is to apply to all authorization IDs.	G
PLANNAME	VARCHAR(24) NOT NULL WITH DEFAULT	Plan name associated with the SQL request. Blank (the default) indicates that the MODENAME specified for the row is to apply to all plan names.	G
LUNAME	VARCHAR(24) NOT NULL	LU name associated with the SQL request.	G
MODENAME	VARCHAR(24) NOT NULL	Name of the logon mode in the VTAM logon mode table to be used in support of the outgoing SQL request. If blank, IBMDB2LM is used for DB2 private protocol connections and IBMRDB is used for DRDA connections.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

SYSIBM.SYSAUDITPOLICIES table

The SYSIBM.SYSAUDITPOLICIES table contains one row for each audit policy.

A user with SECADM authority has the privilege to select from, insert, update, or delete from this catalog table. A user with SQLADM, system DBADM, DATAACCESS, ACCESSCTRL, SYSCTRL or SYSADM authority has the privilege to select from this catalog table.

Column name	Data type	Description	Use
AUDITPOLICYNAME	VARCHAR(128) NOT NULL	Name of the audit policy. The name must be an identifier of 1 to 128 characters and must begin with a letter. Any other values result in an error being returned when audit policy is started.	G
OBJECTSCHEMA	VARCHAR(128) NOT NULL WITH DEFAULT	Schema of the audited object. The object schema only applies to categories, OBJMAINT and EXECUTE.	G
OBJECTNAME	VARCHAR(128) NOT NULL WITH DEFAULT	Name of the object. The object name only applies to categories, OBJMAINT and EXECUTE. Object name can be specified using an SQL LIKE predicate. If the object name is specified using an SQL LIKE predicate, it has to be specified as a delimited identifier. The escape character to be used for the SQL LIKE predicate is obtained from RGFESCP subsystem parameter. If not specified, the default escape character is '+'.	G
OBJECTTYPE	CHAR(1) NOT NULL WITH DEFAULT	Type of the object. **C** Clone table **P** Implicit table created for XML columns **T** Table **blank** All of the above object types **All other values** Error when audit policy is started The object type only applies to categories, OBJMAINT and EXECUTE	G
CREATEDTS	TIMESTAMP NOT NULL WITH DEFAULT	The time when the row was inserted.	G
ALTEREDTS	TIMESTAMP NOT NULL WITH DEFAULT	The time when the row was last updated.	G
CHECKING	CHAR(1) NOT NULL WITH DEFAULT	Indicates if authorization and authentication failures are audited: **A** Audit all failures (Authorization and authentication failures) **blank** Audit none **All other values** Error when audit policy is started	G

Column name	Data type	Description	Use
VALIDATE	CHAR(1) NOT NULL WITH DEFAULT	Indicates if auditing is enabled for when a trusted connection is established or used by a different user: **A** Audit all **blank** Audit none **All other values** Error when audit policy is started	G
OBJMAINT	CHAR(1) NOT NULL WITH DEFAULT	Indicates if auditing is enabled for when the table that is identified by OBJECTSCHEMA, OBJECTNAME, and OBJECTTYPE columns is altered or dropped: **A** Audit when the specified table is altered or dropped **blank** Audit none **All other values** Error when audit policy is started	G
EXECUTE	CHAR(1) NOT NULL WITH DEFAULT	Indicates if auditing is enabled for when the table identified that is by the OBJECTSCHEMA, OBJECTNAME, and OBJECTTYPE columns is accessed during the first operation performed by each unit of work. Also, records bind time information about SQL statements that involve tables that are identified by the OBJECTSCHEMA, OBJECTNAME, and OBJECTTYPE. **A** Audit when the specified table is accessed during the first operation of any kind performed by each unit of work of a utility or application process. **C** Audit when the specified table is accessed during the first insert, update, or delete operation performed by each unit of work. **blank** Audit none **All other values** Error when audit policy is started	G
CONTEXT	CHAR(1) NOT NULL WITH DEFAULT	Indicates if auditing is enabled for the start of a utility, a change to a utility object or phase, and the end of utility: **A** Audit all utilities **blank** Audit none **All other values** Error when audit policy is started	G
SECMAINT	CHAR(1) NOT NULL WITH DEFAULT	Indicates if auditing is enabled for when a grant or revoke is made or a trusted context is created or altered: **A** Audit all **blank** Audit none **All other values** Error when audit policy is started	G

Column name	Data type	Description	Use
SYSADMIN	VARCHAR(128) NOT NULL WITH DEFAULT	Indicates if auditing is enabled for when an operation is performed using an administrative authority to perform system administration tasks: **blank** Audit none * Audit all the authorities **I** Installation SYSADM **L** SYSCTRL **O** SYSOPR **R** Installation SYSOPR **S** SYSADM **All other values** Error when audit policy is started The value of SYSADMIN can be a concatenated string of all supported values. For example, 'LOS' would indicate auditing of any operation that is performed using the administrative authorities: SYSCTRL, SYSOPR, and SYSADM. Multiple occurrences of a value are ignored.	G
DBADMIN	VARCHAR(128) NOT NULL WITH DEFAULT	Indicates if auditing is enabled for when an operation is performed using an administrative authority to perform database administration tasks: **blank** Audit none * Audit all the authorities **B** System DBADM **C** DBCTRL **D** DBADM **E** SECADM **G** ACCESSCTRL **K** SQLADM **M** DBMAINT **P** PACKADM **T** DATAACCESS **All other values** Error when audit policy is started The value of DBADMIN can be a concatenated string of all supported values. For example, 'BMP' would indicate auditing of any operation that is performed using the administrative authorities: System DBADM, DBMAINT, and PACKADM. Multiple occurrences of a value are ignored.	G
DBNAME	VARCHAR(24) NOT NULL WITH DEFAULT	Database name. The database name can be used to specify the database for auditing DBADM, DBCTRL, and DBMAINT authorities. If the database name is not specified, then all the databases, including implicit databases are audited. If the database name is specified, it only applies to DBADM, DBCTRL, and DBMAINT authorities in category, DBADMIN.	G

Column name	Data type	Description	Use
COLLID	VARCHAR(128) NOT NULL WITH DEFAULT	Name of the package collection. The package collection can be used to specify the collection name for auditing PACKADM authority. If specified, all packages in that collection are audited. If the collection name is not specified, packages in all collections are audited. If the package collection is specified, it only applies to PACKADM authority in category, DBADMIN.	G
DB2START	CHAR(1) NOT NULL WITH DEFAULT	ndicates if audit policies are to be started automatically during DB2 start up. Up to 8 audit policies can be specified. **Y** Audit policy will be started automatically during DB2 startup. **S** Audit policy will be started automatically during DB2 startup. The audit policy can be stopped only by a user with SECADM authority. **N** Audit policy will not be started automatically during DB2 startup.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

SYSIBM.SYSAUTOALERTS table

The SYSIBM.SYSAUTOALERTS table contains one row for each recommendation from autonomic procedures.

Column name	Data type	Description	Use
ALERT_ID	BIGINT NOT NULL GENERATED ALWAYS AS IDENTITY	The ID of the alert described in this row.	G
HISTORY_ENTRY_ID	BIGINT NOT NULL	The ID of the entry in the ADMIN_UTLPROCEDURES_HIST procedure that produced this alert.	G
ACTION	VARCHAR(32) NOT NULL	The type of action requested by this alert.	G
TARGET_QUALIFIER	VARCHAR(128) NOT NULL	The qualifier name of the DB2 object (the database name) to which this alert applies.	G
TARGET_OBJECT	VARCHAR(128) NOT NULL	The name of the DB2 object (the table space name) to which this alert applies.	G
TARGET_PARTITION	SMALLINT NOT NULL	The partition number of the DB2 object to which this alert applies. Zero, if this alert applies to all partitions or if the object is not partitioned.	G
OPTIONS	VARCHAR(4000)	The options that should be specified when the corresponding action is run: **USE PROFILE** Use the options specified in the profile **TABLE** Options only apply for this table **COLUMNS** Options only apply for these columns **SAMPLE** The table space is too big and sampling is allowed	G
CREATEDTS	TIMESTAMP NOT NULL WITH DEFAULT	The timestamp when the alert was issued.	G
DURATION	INTEGER	An estimate of the time, in seconds, that would be needed to run the corresponding action. If this column contains NULL, the execution plan might overwrite the time window.	G
STATUS	VARCHAR(32)	The status of the actual planned task. Valid values are: **OPEN** The alert is not yet resolved **INPROGRESS** The alert execution is in progress **COMPLETED** The alert execution is complete	G
STARTTS	TIMESTAMP	The timestamp for when the alert execution started. This column contains NULL if the task execution has not yet started.	G
ENDTS	TIMESTAMP	The timestamp for when the alert execution ended. This column contains NULL if the task execution has not yet ended.	G

Column name	Data type	Description	Use
RETURN_CODE	INTEGER	The return code that is written directly by the autonomic stored procedure that resolved the alert. This column contains NULL if the alert is not yet resolved, if the autonomic stored procedure failed, or if the autonomic stored procedure does not write any return code. A RETURN_CODE of 0 is expected in case of a successful execution.	G
ERROR_MESSAGE	VARCHAR(1331)	An error message that indicates why the alert was not resolved successfully. This column contains NULL if the alert is not yet resolved, or if the autonomic stored procedure that executes the alert does not write any error message. No ERROR_MESSAGE text is expected in case of a successful execution.	G
OUTPUT	CLOB(2M)	The output that is written directly by the autonomic stored procedure that executes the planned task. This column contains NULL if the task is not yet executed, if the execution failed, or if the autonomic stored procedure does not write any output.	G
ROWID	ROWID NOT NULL GENERATED ALWAYS	The ROWID value for the CLOB column of this table.	G

SYSIBM.SYSAUTOALERTS_OUT table

The SYSIBM.SYSAUTOALERTS_OUT table is an auxiliary table for the OUTPUT column of the SYSIBM.SYSAUTOALERTS table.

Column name	Data type	Description	Use
OUTPUT	CLOB(2M)	The output of the autonomic stored procedure.	G

SYSIBM.SYSAUTORUNS_HIST table

The SYSIBM.SYSAUTORUNS_HIST table contains one row for each time an autonomic procedures has been run.

Column name	Data type	Description	Use
HISTORY_ENTRY_ID	BIGINT NOT NULL GENERATED ALWAYS AS IDENTITY	The ID of the entry in the history table.	G
PROC_NAME	VARCHAR(128) NOT NULL	The name of the autonomic stored procedure that produced this entry.	G
STARTTS	TIMESTAMP	The timestamp when the autonomic stored procedure started.	G
ENDTS	TIMESTAMP	The timestamp when the autonomic stored procedure ended.	G
OUTPUT	CLOB(2M)	The output of the autonomic stored procedure.	G
ERROR_MESSAGE	VARCHAR(1331)	An error message that indicates why the autonomic stored procedure was not successful. No ERROR_MESSAGE text is expected in case of a successful execution.	G
RETURN_CODE	INTEGER	The return code written directly by the autonomic stored procedure. This column contains NULL if the autonomic stored procedure execution failed, or if the autonomic stored procedure does not write any return code. A RETURN_CODE of 0 is expected in case of a successful execution.	G
ROWID	ROWID NOT NULL GENERATED ALWAYS	The ROWID value for the OUTPUT column of this table.	G

SYSIBM.SYSAUTORUNS_HISTOU table

The SYSIBM.SYSAUTORUNS_HISTOU table is an auxiliary table for the OUTPUT column of the SYSIBM.SYSAUTORUNS_HIST table.

Column name	Data type	Description	Use
OUTPUT	CLOB(2M)	The output of the autonomic stored procedure.	G

SYSIBM.SYSAUTOTIMEWINDOWS table

The SYSIBM.SYSAUTOTIMEWINDOWS table contains one row for each time period during which autonomic procedures can be run.

Column name	Data type	Description	Use
WINDOW_ID	BIGINT NOT NULL GENERATED ALWAYS AS IDENTITY	The ID of the time window described in this row.	G
DB2_SSID	CHAR(4)	The DB2 member name on which the planned tasks have to be run. If this column contains NULL, the tasks in this time window can be run on any DB2 member.	G
MONTH_WEEK	CHAR(1) NOT NULL	Indicates how the value of the DAY column is interpreted: **M** The value of the DAY column is interpreted as a day of the month **W** The value of the DAY column is interpreted as a day of the week	G
MONTH	INTEGER	Month in which the time window applies. The value will be from 1 (January) to 12 (December). If this column contains NULL, the time window applies to all months. If MONTH_WEEK is 'W', this column must be NULL.	G
DAY	INTEGER	Day of the month or day of the week for which the time window applies. If this column contains NULL, the time window applies to every day of the month or to every day of the week (depending on the value of the MONTH_WEEK column).	G
FROM_TIME	TIME	The time of day at which the time window begins. If this column contains NULL, no limitation on the time exists. This column will contain NULL if the TO_TIME column contains NULL.	G
TO_TIME	TIME	The time of day at which the time window ends. If this column contains NULL, no limitation on the time exists. This column will contain NULL if the FROM_TIME column contains NULL.	G
ACTION	VARCHAR(256)	The comma-separated list of actions that are allowed during this time window. If this column contains NULL, all actions are allowed.	G
MAX_TASKS	INTEGER	The number of concurrent actions that are allowed during this time window. If this column contains NULL, any number of actions are allowed concurrently.	G

SYSIBM.SYSAUXRELS table

The SYSIBM.SYSAUXRELS table contains one row for each auxiliary table created for a LOB column. A base table space that is partitioned must have one auxiliary table for each partition of each LOB column.

Column name	Data type	Description	Use
TBOWNER	VARCHAR(128) NOT NULL	The schema of the base table.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the base table.	G
COLNAME	VARCHAR(128) NOT NULL	Name of the LOB column in the base table.	G
PARTITION	SMALLINT NOT NULL	Partition number if the base table space is partitioned. Otherwise, the value is 0.	G
AUXTBOWNER	VARCHAR(128) NOT NULL	The schema of the auxiliary table.	G
AUXTBNAME	VARCHAR(128) NOT NULL	Name of the auxiliary table.	G
AUXRELOBID	INTEGER NOT NULL	Internal identifier of the relationship between the base table and the auxiliary table.	S
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies. RELCREATED should be used instead.	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. Blank if created prior to Version 9. See Release dependency indicators for all other values.	G

SYSIBM.SYSCHECKDEP table

The SYSIBM.SYSCHECKDEP table contains one row for each reference to a column in a check constraint.

Column name	Data type	Description	Use
TBOWNER	VARCHAR(128) NOT NULL	The schema of the table on which the check constraint is defined.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table on which the check constraint is defined.	G
CHECKNAME	VARCHAR(128) NOT NULL	Name of the check constraint.	G
COLNAME	VARCHAR(128) NOT NULL	Name of the column that the check constraint refers to.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

SYSIBM.SYSCHECKS table

The SYSIBM.SYSCHECKS table contains one row for each check constraint.

Column name	Data type	Description	Use
TBOWNER	VARCHAR(128) NOT NULL	The schema of the table on which the check constraint is defined.	G
CREATOR	VARCHAR(128) NOT NULL	Authorization ID of the creator of the check constraint.	G
DBID	SMALLINT NOT NULL	Internal identifier of the database for the check constraint.	S
OBID	SMALLINT NOT NULL	Internal identifier of the check constraint.	S
TIMESTAMP	TIMESTAMP NOT NULL	Time when the check constraint was created.	G
RBA	CHAR(10) NOT NULL FOR BIT DATA	The log RBA when the check constraint was created.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table on which the check constraint is defined.	G
CHECKNAME	VARCHAR(128) NOT NULL	Check constraint name.	G
CHECKCONDITION	VARCHAR(7400) NOT NULL	Text of the check constraint.	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. Blank if created prior to Version 9. See Release dependency indicators for all other values. The value in this field is not a reliable indicator of release dependencies. RELCREATED should be used instead.	G
ENVID	INTEGER NOT NULL WITH DEFAULT	Internal environment identifier.	G
PERIOD	CHAR(1) NOT NULL WITH DEFAULT	The type of period associated with the check constraint: **B** BUSINESS_TIME check constraint **S** SYSTEM_TIME check constraint **blank** Not applicable	G

SYSIBM.SYSCHECKS2 table

The SYSIBM.SYSCHECKS2 table contains one row for each check constraint for catalog tables created in or after Version 7. Check constraints for catalog tables created before Version 7 are not included in this table.

Column name	Data type	Description	Use
TBOWNER	VARCHAR(128) NOT NULL	The schema of the table on which the check constraint is defined.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table on which the check constraint is defined.	G
CHECKNAME	VARCHAR(128) NOT NULL	Check constraint name.	G
PATHSCHEMAS	VARCHAR(2048) NOT NULL	SQL path at the time the check constraint was created. The path is used to resolve unqualified cast function names that are used in the constraint definition.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies. RELCREATED should be used instead.	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. Blank if created prior to Version 9. See Release dependency indicators for all other values.	G

SYSIBM.SYSCOLAUTH table

The SYSIBM.SYSCOLAUTH table records the UPDATE or REFERENCES privileges that are held by users on individual columns of a table or view.

Column name	Data type	Description	Use
GRANTOR	VARCHAR(128) NOT NULL	Authorization ID or role of the user who granted the privileges. Could also be PUBLIC .	G
GRANTEE	VARCHAR(128) NOT NULL	Authorization ID or role of the user who holds the privilege or the name of an application plan or package that uses the privilege. PUBLIC for a grant to PUBLIC.	G
GRANTEETYPE	CHAR(1) NOT NULL	Type of grantee: **blank** An authorization ID **L** Role **P** An application plan or a package. The grantee is a package if COLLID is not blank.	G
CREATOR	VARCHAR(128) NOT NULL	The schema of the table or view on which the update privilege is held.	G
TNAME	VARCHAR(128) NOT NULL	Name of the table or view.	G
	CHAR(12) NOT NULL	Internal use only	I
	CHAR(6) NOT NULL	Not used	N
	CHAR(8) NOT NULL	Not used	N
COLNAME	VARCHAR(128) NOT NULL	Name of the column to which the UPDATE privilege applies.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
	VARCHAR(128) NOT NULL WITH DEFAULT	Not used	N
COLLID	VARCHAR(128) NOT NULL WITH DEFAULT	If GRANTEE is a package, its collection name. Otherwise, the value is blank.	G
CONTOKEN	CHAR(8) NOT NULL WITH DEFAULT FOR BIT DATA	If GRANTEE is a package, the consistency token of the DBRM from which the package was derived. Otherwise, the value is blank.	S

Column name	Data type	Description	Use
PRIVILEGE	CHAR(1) NOT NULL WITH DEFAULT	Indicates which privilege this row describes: **R** Row pertains to the REFERENCES privilege. **blank** Row pertains to the UPDATE privilege.	G
GRANTEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the GRANT statement was executed.	G
GRANTORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of grantor: **L** Role **blank** Authorization ID that is not a role	G

SYSIBM.SYSCOLDIST table

The SYSIBM.SYSCOLDIST table contains one or more rows for the cardinality, frequency, and histogram statistics for a single column or a column group.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
	SMALLINT NOT NULL	Not used	N
STATSTIME	TIMESTAMP NOT NULL WITH DEFAULT	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
TBOWNER	VARCHAR(128) NOT NULL	The schema of the table that contains the column.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table that contains the column.	G
NAME	VARCHAR(128) NOT NULL	Name of the column. If NUMCOLUMNS is greater than 1, this name identifies the first column name of the set of columns associated with the statistics.	G
COLVALUE	VARCHAR(2000) NOT NULL OR BIT DATA	Contains the data of a frequently occurring value. Statistics are not collected for an index on a ROWID column. If the value has a non-character data type, the data might not be printable.	S
TYPE	CHAR(1) NOT NULL WITH DEFAULT 'F'	The type of statistics gathered: **C** Cardinality **F** Frequent value **H** Histogram Statistics **N** Non-padded frequent value	G
CARDF	FLOAT NOT NULL WITH DEFAULT -1	For TYPE='C', the number of distinct values for the column group. For TYPE='H', the number of distinct values for the column group in a quantile indicated by QUANTILENO.	S
COLGROUPCOLNO	VARCHAR(254) NOT NULL WITH DEFAULT FOR BIT DATA	Identifies the set of columns associated with the statistics. If the statistics are only associated with a single column, the field contains a zero length. Otherwise, the field is an array of SMALLINT column numbers with a dimension equal to the value in NUMCOLUMNS. This is an updatable column.	S
NUMCOLUMNS	SMALLINT NOT NULL WITH DEFAULT 1	Identifies the number of columns associated with the statistics.	G

Column name	Data type	Description	Use
FREQUENCYF	FLOAT NOT NULL WITH DEFAULT -1	Gives the percentage of rows in the table with the value specified in COLVALUE when the number is multiplied by 100. For example, a value of '1' indicates 100%. A value of '.153' indicates 15.3%. When TYPE='H', this is the percentage of rows in table which falls at the quantile indicated by QUANTILENO whose range is limited by [LOWVALUE, HIGHVALUE]. Statistics are not collected for an index on a ROWID column.	G
QUANTILENO	SMALLINT NOT NULL WITH DEFAULT -1	Ordinary sequence number of a quantile in the whole consecutive value range, from low to high. This column is not updatable.	G
LOWVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	For TYPE='H', this is the lower bound for the quantile indicated by QUANTILENO. Not used if TYPE is not 'H'. This column is not updatable.	G
HIGHVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	For TYPE='H', this is the higher bound for the quantile indicated by QUANTILENO. Not used if TYPE is not 'H'. This column is not updatable.	G

SYSIBM.SYSCOLDISTSTATS table

The SYSIBM.SYSCOLDISTSTATS table contains zero or more rows per partition for the cardinality, frequency, and histogram statistics for a single column or a column group.

No row is inserted if the index is a non-partitioned index. Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
	SMALLINT NOT NULL	Not used	N
STATSTIME	TIMESTAMP NOT NULL WITH DEFAULT	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
PARTITION	SMALLINT NOT NULL	Partition number for the table space that contains the table in which the column is defined.	G
TBOWNER	VARCHAR(128) NOT NULL	The schema of the table that contains the column.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table that contains the column.	G
NAME	VARCHAR(128) NOT NULL	Name of the column. If NUMCOLUMNS is greater than 1, this name identifies the first column name of the set of columns associated with the statistics.	G
COLVALUE	VARCHAR(2000) NOT NULL FOR BIT DATA	Contains the data of a frequently occurring value. Statistics are not collected for an index on a ROWID column. If the value has a non-character data type, the data might not be printable.	S
TYPE	CHAR(1) NOT NULL WITH DEFAULT 'F'	The type of statistics gathered: **C** Cardinality **F** Frequent value **H** Histogram statistics **N** Non-padded frequent value	G
CARDF	FLOAT NOT NULL WITH DEFAULT -1	If TYPE is C, the value is the number of distinct values for the column group. If TYPE is N or TYPE is F, the value is the number of rows or keys in the partition for which the FREQUENCYF value applies. If TYPE is H, the number of distinct values for the column group in a quantile indicated by QUANTILENO.	S

Column name	Data type	Description	Use
COLGROUPCOLNO	VARCHAR(254) NOT NULL WITH DEFAULT FOR BIT DATA	Identifies the set of columns associated with the statistics. If the statistics are only associated with a single column, the field contains a zero length. Otherwise, the field is an array of SMALLINT column numbers with a dimension equal to the value in NUMCOLUMNS. This is an updatable column.	S
NUMCOLUMNS	SMALLINT NOT NULL WITH DEFAULT 1	Identifies the number of columns associated with the statistics.	G
FREQUENCYF	FLOAT NOT NULL WITH DEFAULT -1	Gives the percentage of rows in the table with the value specified in COLVALUE when the number is multiplied by 100. For example, a value of '1' indicates 100%. A value of '.153' indicates 15.3%. When TYPE='H', this is the percentage of rows in table which falls in the quantile indicated by QUANTILENO whose range is limited by [LOWVALUE, HIGHVALUE]. Statistics are not collected for an index on a ROWID column.	G
	VARCHAR(1000) NOT NULL WITH DEFAULT FOR BIT DATA	Internal use only	I
QUANTILENO	SMALLINT NOT NULL WITH DEFAULT -1	Ordinary sequence number of a quantile in the whole consecutive value range, from low to high. This column is not updatable.	G
LOWVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	For TYPE='H', this is the lower bound for the quantile indicated by QUANTILENO. Not used if TYPE is not 'H'. This column is not updatable.	G
HIGHVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	For TYPE='H', this is the higher bound for the quantile indicated by QUANTILENO. Not used if TYPE is not 'H'. This column is not updatable.	G

SYSIBM.SYSCOLDIST_HIST table

The SYSIBM.SYSCOLDIST_HIST table contains rows from SYSCOLDIST.

Rows are added or changed in this table when RUNSTATS collects history statistics. Rows in this table can also be inserted, updated, and deleted.

Column name	Data type	Description	Use
STATSTIME	TIMESTAMP NOT NULL	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics.	G
TBOWNER	VARCHAR(128) NOT NULL	The schema of the table that contains the column.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table that contains the column.	G
NAME	VARCHAR(128) NOT NULL	Name of the column. If NUMCOLUMNS is greater than 1, this name identifies the first column name of the set of columns associated with the statistics.	G
COLVALUE	VARCHAR(2000) NOT NULL FOR BIT DATA	Contains the data of a frequently occurring value. Statistics are not collected for an index on a ROWID column. If the value has a non-character data type, the data might not be printable.	S
TYPE	CHAR(1) NOT NULL WITH DEFAULT 'F'	The type of statistics gathered: **C** Cardinality **F** Frequent value **H** Histogram Statistics **N** Non-padded frequent value	G
CARDF	FLOAT(8) NOT NULL WITH DEFAULT -1	When TYPE='C', this is the number of distinct values for the column group. When TYPE='H', this is he number of distinct values for the column group in a quantile indicated by QUANTILENO. The value is -1 if statistics have not been gathered.	S
COLGROUPCOLNO	VARCHAR(254) NOT NULL FOR BIT DATA	Identifies the set of columns associated with the statistics. If the statistics are only associated with a single column, the field contains a zero length. Otherwise, the field is an array of SMALLINT column numbers with a dimension equal to the value in NUMCOLUMNS.	S
NUMCOLUMNS	SMALLINT NOT NULL WITH DEFAULT 1	Identifies the number of columns associated with the statistics.	G

Column name	Data type	Description	Use
FREQUENCYF	FLOAT(8) NOT NULL DEFAULT -1	Gives the percentage of rows in the table with the value specified in COLVALUE when the number is multiplied by 100. For example, a value of '1' indicates 100%. A value of '.153' indicates 15.3%. When TYPE='H', this is the percentage of rows in table which falls in the quantile indicated by QUANTILENO whose range is limited by [LOWVALUE, HIGHVALUE]. Statistics are not collected for an index on a ROWID column. The value is -1 if statistics have not been gathered.	G
IBMREQD	CHAR(1) NOT NULL DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
QUANTILENO	SMALLINT NOT NULL WITH DEFAULT -1	Ordinary sequence number of a quantile in the whole consecutive value range, from low to high. This column is not updatable.	G
LOWVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	For TYPE='H', this is the lower bound for the quantile indicated by QUANTILENO. Not used if TYPE is not 'H'. This column is not updatable.	G
HIGHVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	For TYPE='H', this is the higher bound for the quantile indicated by QUANTILENO. Not used if TYPE is not 'H'. This column is not updatable.	G

SYSIBM.SYSCOLSTATS table

The SYSIBM.SYSCOLSTATS table contains partition statistics for selected columns. For each column, a row exists for each partition in the table.

Rows are inserted when RUNSTATS collects either indexed column statistics or non-indexed column statistics for a partitioned table space. No row is inserted if the table space is nonpartitioned. Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
HIGHKEY	VARCHAR(2000) NOT NULL FOR BIT DATA	Highest value of the column within the partition. Blank if statistics have not been gathered or the column is an indicator column, a node ID column, or a column of an XML table. If the column has a non-character data type, the data might not be printable. If the partition is empty, the value is a string of length 0.	S
HIGH2KEY	VARCHAR(2000) NOT NULL FOR BIT DATA	Second highest value of the column within the partition. Blank if statistics have not been gathered or the column is an indicator column, a node ID column, or a column of an XML table. If the column has a non-character data type, the data might not be printable. If the partition is empty, the value is a string of length 0.	S
LOWKEY	VARCHAR(2000) NOT NULL FOR BIT DATA	Lowest value of the column within the partition. Blank if statistics have not been gathered or the column is an indicator column, a node ID column, or a column of an XML table. If the column has a non-character data type, the data might not be printable. If the partition is empty, the value is a string of length 0.	S
LOW2KEY	VARCHAR(2000) NOT NULL FOR BIT DATA	Second lowest value of the column within the partition. Blank if statistics have not been gathered or the column is an indicator column, a node ID column, or a column of an XML table. If the column has a non-character data type, the data might not be printable. If the partition is empty, the value is a string of length 0.	S
COLCARD	INTEGER NOT NULL	Number of distinct column values in the partition.	S
STATSTIME	TIMESTAMP NOT NULL	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics. If the value is '0001-01-02.00.00.00.000000', which indicates that an ALTER TABLE statement was executed to change the length of a VARCHAR column, RUNSTATS should be run to update the statistics before they are used.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
PARTITION	SMALLINT NOT NULL	Partition number for the table space that contains the table in which the column is defined.	G

Column name	Data type	Description	Use
TBOWNER	VARCHAR(128) NOT NULL	Schema or qualifier of the table that contains the column.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table that contains the column.	G
NAME	VARCHAR(128) NOT NULL	Name of the column.	G
	VARCHAR(1000) NOT NULL FOR BIT DATA	Internal use only	I
STATS_FORMAT	CHAR(1) NOT NULL WITH DEFAULT	The type of statistics gathered: **blank** Statistics have not been collected or varchar column statistical values are padded. **N** Varchar column statistical values are not padded. This is an updatable column.	G

SYSIBM.SYSCOLUMNS table

The SYSIBM.SYSCOLUMNS table contains one row for every column of each table and view.

Column name	Data type	Description	Use
NAME	VARCHAR(128) NOT NULL	Name of the column.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table or view which contains the column.	G
TBCREATOR	VARCHAR(128) NOT NULL	The schema of the table or view that contains the column.	G
COLNO	SMALLINT NOT NULL	Numeric place of the column in the table or view; for example 4 (out of 10).	G

Column name	Data type	Description	Use
COLTYPE	CHAR(8) NOT NULL	The type of the column specified in the definition of the column: **INTEGER** Large integer **SMALLINT** Small integer **FLOAT** Floating-point **CHAR** Fixed-length character string **VARCHAR** Varying-length character string **LONGVAR** Varying-length character string (for columns that were added before Version 9) **DECIMAL** Decimal **GRAPHIC** Fixed-length graphic string **VARG** Varying-length graphic string **LONGVARG** Varying-length graphic string (for columns that were added before Version 9) **DATE** Date **TIME** Time **TIMESTMP** Timestamp **TIMESTZ** Timestamp with time zone **BLOB** Binary large object **CLOB** Character large object **DBCLOB** Double-byte character large object **ROWID** Row ID data type **DISTINCT** Distinct type **XML** XML data type **BIGINT** Big integer **BINARY** Fixed-length binary string **VARBIN** Varying-length binary string **DECFLOAT** Decimal floating point	G

Column name	Data type	Description	Use
LENGTH	SMALLINT NOT NULL	Length attribute of the column or, in the case of a decimal column, its precision. The number does not include the internal prefixes that are used to record the actual length and null state, where applicable. **INTEGER** 4 **SMALLINT** 2 **BIGINT** 8 **FLOAT** 4 or 8 **CHAR** Length of string **VARCHAR** Maximum length of string **LONGVAR** Maximum length of string (for columns that were added before Version 9) **DECIMAL** Precision of number **DECFLOAT** 8 or 16 **GRAPHIC** Number of DBCS characters **VARGRAPHIC** Maximum number of DBCS characters **LONGVARG** Maximum number of DBCS characters (for columns that were added before Version 9) **BINARY** Length of string **VARBINARY** Maximum length of string **DATE** 4 **TIME** 3 **TIMESTAMP WITHOUT TIME ZONE** The integral part of ((p+1)/2) + 7 where p is the precision of the timestamp **TIMESTAMP WITH TIME ZONE** The integral part of ((p+1)/2) + 9 where p is the precision of the timestamp	G

Column name	Data type	Description	Use
LENGTH (continued)	SMALLINT NOT NULL	**LOB** 4 - For a table, a field of length of 4 is stored in the base table. The maximum length of the LOB column is found in LENGTH2. **INLINE LOB** Greater than 4 - For a table, a field of length 4 plus the inline length (in byte) is stored in the base table. The maximum length of the LOB column is found in LENGTH2. **BLOB** 4 - For a table, a field of length of 4 is stored in the base table. The maximum length of the LOB column is found in LENGTH2. **CLOB** 4 - For a table, a field of length of 4 is stored in the base table. The maximum length of the CLOB column is found in LENGTH2. **DBCLOB** 4 - For a table, a field of length of 4 is stored in the base table. The maximum length of the DBCLOB column is found in LENGTH2. **ROWID** 17 - The maximum length of the stored portion of the identifier. **XML** 6 **DISTINCT** The length of the source data type.	G
SCALE	SMALLINT NOT NULL	If the column type is DECIMAL, this value represents the scale. If the column type is timestamp or timestamp with time zone, this value represents the number of fractional second digits. Otherwise the value is 0. If the column is a timestamp type, the LENGTH is 10 and the SCALE is 0, the number of fractional second digits is 6.	G
NULLS	CHAR(1) NOT NULL	Whether the column can contain null values: **N** No **Y** Yes The value can be N for a view column that is derived from an expression that is not a simple column name or constant, or from a function. Nevertheless, such a column allows nulls when an outer select list refers to it.	G
COLCARD	INTEGER NOT NULL	Not used	N
HIGH2KEY	VARCHAR(2000) NOT NULL FOR BIT DATA	Second highest value of the column. Blank if statistics have not been gathered, or the column is an indicator column or a column of an auxiliary table. If the column has a non-character data type, the data might not be printable. If the table is empty, the value is a string of length 0. This is an updatable column.	S

Column name	Data type	Description	Use
LOW2KEY	VARCHAR(2000) NOT NULL FOR BIT DATA	Second lowest value of the column. Blank if statistics have not been gathered, or the column is an indicator column or a column of an auxiliary table. If the column has a non-character data type, the data might not be printable. If the table is empty, the value is a string of length 0. This is an updatable column.	S
UPDATES	CHAR(1) NOT NULL	Whether the column can be updated: **N** No **Y** Yes The value is N if the column is: • Derived from a function or expression • A column with a row ID data type (or a distinct type based on a row ID type) • A read-only view	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
REMARKS	VARCHAR(762) NOT NULL	A character string provided by the user with the COMMENT statement.	G

Column name	Data type	Description	Use
DEFAULT	CHAR(1) NOT NULL	The contents of this column are meaningful only if the TYPE column for the associated SYSTABLES row indicates that this is for a table (T) or a created temporary table (G). Default indicator: **A** The column has a row ID data type (COLTYPE='ROWID') and the GENERATED ALWAYS attribute. **B** The column has a default value that depends on the data type of the column. **Data type** **Default Value** **Numeric** 0 **Fixed-length character or graphic string** Blanks **Fixed-length binary string** Hexadecimal zeros **Varying-length string** A string length of 0 **Date** The current date **Time** The current time **Timestamp** The current timestamp **Timestamp with time zone** The current timestamp with time zone **D** The column has a row ID data type (COLTYPE='ROWID') and the GENERATED BY DEFAULT attribute. **E** The column is defined with the FOR EACH ROW ON UPDATE and GENERATED ALWAYS attributes. **F** The column is defined with the FOR EACH ROW ON UPDATE and GENERATED BY DEFAULT attributes. **I** The column is defined with the AS IDENTITY and GENERATED ALWAYS attributes. **J** The column is defined with the AS IDENTITY and GENERATED BY DEFAULT attributes. **K** The column is defined for the implicit DOCID column for a base table that contains XML data. **L** The column is defined with the AS SECURITY LABEL attribute. **N** The column has no default value. **Q** The column is defined with the AS ROW BEGIN attribute. **R** The column is defined with the AS ROW END attribute.	G

Column name	Data type	Description	Use
DEFAULT (continued)	CHAR(1) NOT NULL	Default indicator: **S** The column has a default value that is the value of the SQL authorization ID of the process at the time a default value is used. **U** The column has a default value that is the value of the SESSION_USER special register at the time a default value is used. **Y** If the NULLS column is Y, the column has a default value of null. If the NULLS column is N, the default value depends on the data type of the column. **Data type** — **Default Value** **Numeric** — 0 **Fixed-length character string** — Blanks **Fixed-length graphic string** — Blanks **Fixed-length binary string** — Hexadecimal blanks **Varying-length string** — A string length of 0 **Date** — The current date **Time** — The current time **Timestamp** — The current timestamp **Timestamp with time zone** — The current timestamp with time zone **X** The column is defined with the AS TRANSACTION START ID attribute. **1** The column has a default value that is the string constant found in the DEFAULTVALUE column of this table row. The column has a graphic data type and has a default value that is the graphic string found in the DEFAULTVALUE column of this table row. **2** The column has a default value that is the floating-point constant found in the DEFAULTVALUE column of this table row. **3** The column has a default value that is the decimal constant found in the DEFAULTVALUE column of this table row. **4** The column has a default value that is the integer constant found in the DEFAULTVALUE column of this table row. **5** The column has a default value that is the hexadecimal character string found in the DEFAULTVALUE column of this table row.	G

Column name	Data type	Description	Use
DEFAULT (continued)	CHAR(1) NOT NULL	Default indicator: **6** The column has a default value that is the UX string found in the DEFAULTVALUE column of this table row. **7** The column has a graphic data type and has a default value that is the character string constant found in the DEFAULTVALUE column of this table row. **8** The column has a character data type and has a default value that is the graphic string constant found in the DEFAULTVALUE column of this table row. **9** The column has a default value that is the DECFLOAT constant found in the DEFAULTVALUE column of this table row.	G
KEYSEQ	SMALLINT NOT NULL	The numeric position of the column within the primary key of the table. The value is 0 if it is not part of a primary key.	G
FOREIGNKEY	CHAR(1) NOT NULL	Applies to character or CLOB columns, where it indicates the subtype of the data: **B** BIT data **M** MIXED data **S** SBCS data **blank** Indicates one of the following subtypes: • MIXED data if the encoding scheme is UNICODE, or if the encoding scheme is not UNICODE and the value of MIXED DATA on installation panel DSNTIPS is YES • SBCS data if the encoding scheme is not UNICODE and the value of MIXED DATA on the installation panel DSNTIPS is NO. For views defined prior to Version 7, subtype information is not available and the default (MIXED or SBCS) is used.	G
FLDPROC	CHAR(1) NOT NULL	Whether the column has a field procedure: **N** No **Y** Yes **blank** The column is for a view defined prior to Version 7. Views defined after Version 7 contain Y or N.	G
LABEL	VARCHAR(90) NOT NULL	The column label provided by the user with a LABEL statement; otherwise, the value is an empty string.	G
STATSTIME	TIMESTAMP NOT NULL WITH DEFAULT	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.00.000000'. If the value is '0001-01-02.00.00.00.000000', which indicates that an ALTER TABLE statement was executed to change the length of a VARCHAR column, RUNSTATS should be run to update the statistics before they are used. This is an updatable column.	G

Column name	Data type	Description	Use
DEFAULTVALUE	VARCHAR(1536) NOT NULL WITH DEFAULT	This field is meaningful only if the column being described is for a table (the TYPE column of the associated SYSTABLES row is T for table or G for created temporary table). When the DEFAULT column is 1, 2, 3, 4, 5, 6, 7, 8, or 9, this field contains the default value of the column. If the default value is a string constant or a hexadecimal constant (DEFAULT is 1, 5, 6, 7, or 8 respectively), the value is stored without delimiters. If the default value is a numeric constant (DEFAULT is 2, 3, 4, or 9), the value is stored as specified by the user, including sign and decimal point representation, or special constant values, as appropriate for the constant. When the DEFAULT column is S or U and the default value was specified when a new column was defined with the ALTER TABLE statement, this field contains the value of the CURRENT SQLID or SESSION_USER special register at the time the ALTER TABLE statement was executed. Remember that this default value applies only to rows that existed before the ALTER TABLE statement was executed. When the DEFAULT column is L and the column was added as a new column with the ALTER TABLE statement, this field contains the security label of the user at the time the ALTER TABLE statement was executed. Remember that this default value applies only to rows that existed before the ALTER TABLE statement was executed.	G
COLCARDF	FLOAT NOT NULL WITH DEFAULT	Estimated number of distinct values in the column. For an indicator column, this is the number of LOBs that are not null and have a length greater than zero. The value is -1 if statistics have not been gathered. The value is -2 if the column is a LOB column. This is an updatable column.	S
COLSTATUS	CHAR(1) NOT NULL WITH DEFAULT	Indicates the status of the definition of a column: **I** The definition is incomplete because a LOB table space, auxiliary table, or index on an auxiliary table has not been created for the column. **blank** The definition is complete.	G
LENGTH2	INTEGER NOT NULL WITH DEFAULT	Maximum length of the data retrieved from the column. Possible values are: **0** Column is not a LOB or ROWID column **40** For a ROWID column, the length of the returned value **1 to 2,147,483,647 bytes** For a LOB column, the maximum length	G
DATATYPEID	INTEGER NOT NULL WITH DEFAULT	For a built-in data type, the internal ID of the built-in type. For a distinct type, the internal ID of the distinct type. If the column was created prior to Version 6, the value is 0.	S

Column name	Data type	Description	Use
SOURCETYPEID	INTEGER NOT NULL WITH DEFAULT	For a built-in data type, 0. For a distinct type, the internal ID of the built-in data type upon which the distinct type is based. If the column was created prior to Version 6, the value is 0.	S
TYPESCHEMA	VARCHAR(128) NOT NULL WITH DEFAULT 'SYSIBM'	If COLTYPE is 'DISTINCT', the schema of the distinct type. Otherwise, the value is 'SYSIBM'.	G
TYPENAME	VARCHAR(128) NOT NULL WITH DEFAULT	If COLTYPE is 'DISTINCT', the name of the distinct type. Otherwise, the value is the same as the value of the COLTYPE column. TYPENAME is set only for columns created in Version 6 or later. The value for columns created earlier is not filled in.	G
CREATEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Timestamp when the column was created. The value is '0001-01-01.00.00.00.000000' if the column was created prior to migration to Version 6 or if the column is in a catalog table.	G
STATS_FORMAT	CHAR(1) NOT NULL WITH DEFAULT	The type of statistics gathered: **blank** Statistics have not been collected or varchar column statistical values are padded. **N** Varchar column statistical values are not padded. This is an updatable column.	G
PARTKEY_COLSEQ	SMALLINT NOT NULL WITH DEFAULT	The numeric position of the column within the partitioning key of the table. The value is 0 if it is not part of the partitioning key. This column is applicable only if the table uses table-controlled partitioning.	G
PARTKEY_ORDERING	CHAR(1) NOT NULL WITH DEFAULT	Order of the column in the partitioning key: **A** Ascending **D** Descending **blank** Column is not used as part of a partitioning key This column is applicable only if the table uses table-controlled partitioning.	G
ALTEREDTS	TIMESTAMP NOT NULL WITH DEFAULT	Timestamp when alter occurred.	G
CCSID	INTEGER NOT NULL WITH DEFAULT	CCSID of the column. 0 if the object was created prior to Version 8 or is not a string column. The value is also 0 if the object is not a VARBINARY column defined as a Unicode column.	G
HIDDEN	CHAR(1) NOT NULL WITH DEFAULT 'N'	Indicates whether the column is implicitly hidden: **P** Partially hidden. The column is implicitly hidden from SELECT *. **N** Not hidden. The column is visible to all SQL statements.	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. See Release dependency indicators for the values.	G

Column name	Data type	Description	Use
HASHKEY_COLSEQ	SMALLINT NOT NULL WITH DEFAULT	The column's numeric position within the table's hash key. The value is 0 if the column is not part of the hash key. This column is applicable only if the table that use hash organization.	G
CONTROL_ID	INTEGER NOT NULL WITH DEFAULT	Internal identifier of the column access control mask defined for this column. 0 if no column access control mask is defined for the column.	S
XML_TYPEMOD_ID	INTEGER NOT NULL WITH DEFAULT	The ID of the XML type modifier. It is set to 0 if the column is not an XML column or has no XML type modifier.	G
PERIOD	CHAR(1) NOT NULL WITH DEFAULT	Indicates whether the column is the start or the end of the period for a SYSTEM_TIME or BUSINESS_TIME period: **B** Column is the start of period BUSINESS_TIME. **C** Column is the end of period BUSINESS_TIME. **S** Column is the start of period SYSTEM_TIME. **T** Column is the end of period SYSTEM_TIME. **blank** Column is not used as either the start or the end of a period.	G
GENERATED_ ATTRIBUTE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the columns generated attribute: **A** Column is defined as GENERATED_ALWAYS. **D** Column is defined as GENERATED BY DEFAULT. **blank** Not applicable or the value of the DEFAULT column is A, D, E, F, I, or J or defined from a prior release of DB2.	G

SYSIBM.SYSCOLUMNS_HIST table

The SYSIBM.SYSCOLUMNS_HIST table contains rows from SYSCOLUMNS.

Rows are added or changed in this table when RUNSTATS collects history statistics. Rows in this table can also be inserted, updated, and deleted.

Column name	Data type	Description	Use
NAME	VARCHAR(128) NOT NULL	Name of the column.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table or view that contains the column.	G
TBCREATOR	VARCHAR(128) NOT NULL	Schema or qualifier of the table or view that contains the column.	G
COLNO	SMALLINT NOT NULL	Numeric place of the column in the table or view. For example 4 (out of 10).	G

Column name	Data type	Description	Use
COLTYPE	CHAR(8) NOT NULL	The type of the column specified in the definition of the column: **INTEGER** Large integer **SMALLINT** Small integer **FLOAT** Floating-point **CHAR** Fixed-length character string **VARCHAR** Varying-length character string **LONGVAR** Varying-length character string (for columns that were added before Version 9) **DECIMAL** Decimal **GRAPHIC** Fixed-length graphic string **VARG** Varying-length graphic string **LONGVARG** Varying-length graphic string (for columns that were added before Version 9) **DATE** Date **TIME** Time **TIMESTAMP** Timestamp **TIMESTZ** Timestamp with time zone **BLOB** Binary large object **CLOB** Character large object **DBCLOB** Double-byte character large object **ROWID** Row ID data type **DISTINCT** Distinct type **XML** XML data type **BIGINT** Big integer **BINARY** Fixed-length binary string **VARBIN** Varying-length binary string **DECFLOAT** Decimal floating point	G

Column name	Data type	Description	Use
LENGTH	SMALLINT NOT NULL	Length attribute of the column or, in the case of a decimal column, its precision. The number does not include the internal prefixes that are used to record the actual length and null state, where applicable. **INTEGER** 4 **SMALLINT** 2 **FLOAT** 4 or 8 **CHAR** Length of string **VARCHAR** Maximum length of string **LONGVAR** Maximum length of string (for columns that were added before Version 9) **DECIMAL** Precision of number **GRAPHIC** Number of DBCS characters **VARGRAPHIC** Maximum number of DBCS characters **LONGVARG** Maximum number of DBCS characters (for columns that were added before Version 9) **DATE** 4 **TIME** 3 **TIMESTAMP WITHOUT TIME ZONE** The integral part of ((p+1)/2) + 7 where p is the precision of the timestamp **TIMESTAMP WITH TIME ZONE** The integral part of ((p+1)/2) + 9 where p is the precision of the timestamp **BLOB** 4 - The length of the field that is stored in the base table. The maximum length of the LOB column is found in LENGTH2. **CLOB** 4 - The length of the field that is stored in the base table. The maximum length of the CLOB **DBCLOB** 4 - The length of the field that is stored in the base table. The maximum length of the DBCLOB column is found in LENGTH2. **ROWID** 17 - The maximum length of the stored portion of the identifier. **DISTINCT** The length of the source data type. **XML** 6 **BIGINT** 8 **BINARY** The length of the string **VARBINARY** The maximum length of string **DECFLOAT** 8 or 16	G

Column name	Data type	Description	Use
LENGTH2	INTEGER NOT NULL	Maximum length of the data retrieved from the column. Possible values are: **0** Column is not a LOB or ROWID column **40** For a ROWID column, the length of the returned value **1 to 2 147 483 647 bytes** For a LOB column, the maximum length	G
NULLS	CHAR(1) NOT NULL	Whether the column can contain null values: **N** No **Y** Yes	G
HIGH2KEY	VARCHAR(2000) NOT NULL FOR BIT DATA	Second highest value of the column. Blank if statistics have not been gathered, or the column is an indicator column or a column of an auxiliary table. If the column has a non-character data type, the data might not be printable.	S
LOW2KEY	VARCHAR(2000) NOT NULL FOR BIT DATA	Second lowest value of the column. Blank if statistics have not been gathered, or the column is an indicator column or a column of an auxiliary table. If the column has a non-character data type, the data might not be printable.	S
STATSTIME	TIMESTAMP NOT NULL	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.00.000000'. If the value is '0001-01-02.00.00.00.000000', which indicates that an ALTER TABLE statement was executed to change the length of a VARCHAR column, RUNSTATS should be run to update the statistics before they are used.	G
COLCARDF	FLOAT(8) NOT NULL WITH DEFAULT -1	Estimated number of distinct values in the column. For an indicator column, this is the number of LOBs that are not null and have a length greater than zero. The value is -1 if statistics have not been gathered. The value is -2 if the column is a LOB column.	S
IBMREQD	CHAR(1) NOT NULL DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
STATS_FORMAT	CHAR(1) NOT NULL WITH DEFAULT	The type of statistics gathered: **blank** Statistics have not been collected or varchar column statistical values are padded. **N** Varchar column statistical values are not padded. This is an updatable column.	G

SYSIBM.SYSCONSTDEP table

The SYSIBM.SYSCONSTDEP table records dependencies on check constraints or user-defined defaults for a column.

Column name	Data type	Description	Use
BNAME	VARCHAR(128) NOT NULL	Name of the object on which the dependency exists.	G
BSCHEMA	VARCHAR(128) NOT NULL	Schema of the object on which the dependency exists.	G
BTYPE	CHAR(1) NOT NULL	Type of object on which the dependency exists: **F** Function instance	G
DTBNAME	VARCHAR(128) NOT NULL	Name of the table to which the dependency applies.	G
DTBCREATOR	VARCHAR(128) NOT NULL	The schema of the table to which the dependency applies.	G
DCONSTNAME	VARCHAR(128) NOT NULL	If DTYPE = 'C', the unqualified name of the check constraint. If DTYPE = 'D', a column name.	G
DTYPE	CHAR(1) NOT NULL	Type of object: **C** Check constraint **D** User-defined default constant	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
DTBOWNER	VARCHAR(128) NOT NULL WITH DEFAULT	Authorization ID of the owner of the table or a zero length string for tables that were created in a DB2 release prior to Version 9.	G
OWNERTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of owner: **blank** Authorization ID **R** Role	G

SYSIBM.SYSCONTEXT table

The SYSIBM.SYSCONTEXT table contains one row for each trusted context.

Column name	Data type	Description	Use
NAME	VARCHAR(128) NOT NULL	Name of the trusted context.	G
CONTEXTID	INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY	Internal context ID.	G
DEFINER	VARCHAR(128) NOT NULL	Authorization ID or role that defined the trusted context.	G
DEFINERTYPE	CHAR(1) NOT NULL	The type of the definer: **L** Role **blank** Authorization ID	G
SYSTEMAUTHID	VARCHAR(128) NOT NULL	The DB2 primary authorization ID that is used to establish the connection. For remote requests, SYSTEMAUTHID is derived from the system user ID that is provided by an external entity, such as a middleware server. For local requests, SYSTEMAUTHID depends on one of the following sources of the address space: **BATCH** USER parameter on JOB statement **RRSAF** USER parameter on JOB statement or RACF user **TSO** TSO logon ID	G
DEFAULTROLE	VARCHAR(128) NOT NULL	Name of the trusted context default role.	G
OBJECTOWNERTYPE	CHAR(1) NOT NULL	Whether the ROLE AS OBJECT OWNER AND QUALIFIER clause is specified in the definition of this trusted context: **L** ROLE AS OBJECT OWNER AND QUALIFIER is specified. A role owns any object created in the trusted context. The role is used as the default for the CURRENT SCHEMA special register. The role is included in the SQL PATH. **blank** ROLE AS OBJECT OWNER is not specified. An authorization ID owns any object created in the trusted context.	G
CREATEDTS	TIMESTAMP NOT NULL	The time when the trusted context is created.	G
ALTEREDTS	TIMESTAMP NOT NULL	The time when the trusted context is last altered.	G

Column name	Data type	Description	Use
ENABLED	CHAR(1) NOT NULL	The status of the trusted context: **Y** Enabled **N** Disabled	G
ALLOWPUBLIC	CHAR(1) NOT NULL	Whether the connection is allowed to be reused for PUBLIC: **Y** Connection reuse is allowed **N** Connection reuse is not allowed	G
AUTHENTICATE-PUBLIC	CHAR(1) NOT NULL	Whether authentication is required for PUBLIC when ALLOWPUBLIC is Y: **Y** Authentication token is required for PUBLIC. For local requests, the token is the password. For remote requests, the token can be a password, a RACF passticket, or a KERBEROS token **N** Authentication is not required	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. See Release dependency indicators for the values.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies. RELCREATED should be used instead.	G
REMARKS	VARCHAR(762) NOT NULL	A character string that is provided using the COMMENT statement.	G
DEFAULT-SECURITYLABEL	VARCHAR(24) NOT NULL	Name of the context default RACF security label.	G

SYSIBM.SYSCONTEXTAUTHIDS table

The SYSIBM.SYSCONTEXTAUTHIDS table contains one row for each authorization ID with which the trusted context can be used.

Column name	Data type	Description	Use
CONTEXTID	INTEGER NOT NULL	The internal trusted context ID.	G
AUTHID	VARCHAR(128) NOT NULL	The primary authorization ID that can reuse a connection. When using RACF, this is a RACF profile name that contains the primary authorization IDs that are permitted to use the connection in the identified trusted context.	G
AUTHENTICATE	CHAR(1) NOT NULL	Whether authentication is required for the authorization ID in the AUTHID column: **Y** Authentication token is required for the authorization ID. For local requests, the token is the password. For remote requests, the token can be a password, a RACF passticket, or a Kerberos token **N** Authentication is not required	G
ROLE	VARCHAR(128) NOT NULL	The role for the authorization ID in the AUTHID column. The role supersedes the default role that is defined for the trusted context.	G
CREATEDTS	TIMESTAMP NOT NULL	The time when the authorization ID is added to the trusted context.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
SECURITYLABEL	VARCHAR(24) NOT NULL	RACF security label for AUTHID. The security label supersedes the default security label, if any, that is defined for the context.	G

SYSIBM.SYSCONTROLS table

The SYSIBM.SYSCONTROLS table contains one row for each row permission and column mask.

Column name	Data type	Description	Use
SCHEMA	VARCHAR(128) NOT NULL	Schema of the row permission or column mask.	G
NAME	VARCHAR(128) NOT NULL	Name of the row permission or column mask.	G
OWNER	VARCHAR(128) NOT NULL	Owner of the row permission or column mask.	G
OWNERTYPE	CHAR(1) NOT NULL	Indicates the type of the owner: **blank** An authorization ID **L** Role	G
TBSCHEMA	VARCHAR(128) NOT NULL	Schema of the table for which the row permission or column mask is defined.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table for which the row permission or column mask is defined.	G
TBCORRELATION	VARCHAR(128) NOT NULL WITH DEFAILT	If specified, the correlation name of the table for which the row permission or column mask is defined. Otherwise, the value is an empty string.	G
COLNAME	VARCHAR(128) NOT NULL	Column name for which the column mask is defined. Blank if this is a row permission.	G
COLNO	SMALLINT NOT NULL	Column number for which the column mask is defined. 0 if this is a row permission.	G
CONTROL_ID	INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY	Internal access control ID.	S
CONTROL_TYPE	CHAR(1) NUT NULL	Indicates the type of the access control object: **R** Row permission **M** Column mask	G
ENFORCED	CHAR(1) NUT NULL	Indicates the type of the access enforced by the row permission. Column mask always has a value of 'A'. **A** All access	G
IMPLICIT	CHAR(1) NUT NULL	Indicates whether the row permission was implicitly created: **N** The row permission was explicitly created or this is a column mask **Y** The row permission was implicitly created	G

Column name	Data type	Description	Use
ENABLE	CHAR(1) NUT NULL	Indicates whether the row permission or the column mask is enabled for access control: **N** Not enabled **Y** Enabled	G
STATUS	CHAR(1) NUT NULL	Indicates the status of the row permission or column mask definition: **blank** The definition of the row permission or column mask is complete. **R** An error occurred when an attempt was made to regenerate the row permission or column mask.	G
CREATEDTS	TIMESTAMP NOT NULL	The timestamp when the row permission or column mask was created.	G
RELCREATED	CHAR(1) NUT NULL	The release of DB2 in which the row permission or column mask was created. See Release dependency indicators for values.	G
ALTEREDTS	TIMESTAMP NOT NULL	The timestamp when the row permission or column mask was last changed.	G
REMARKS	VARCHAR(762) NOT NULL	A character string provided by using the COMMENT ON statement.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies. RELCREATED should be used instead.	G
ENVID	INTEGER NOT NULL	Internal identifier of the environment.	G
ROWID	ROWID	Row identifier to support LOB columns in the table.	G
RULETEXT	CLOB(2MB) NOT NULL	The source text of the search condition or expression portion of the CREATE PERMISSION or CREATE MASK statement. **Note:** The lowercase letters in ordinary tokens are folded to uppercase in the text. However, lowercase letters in ordinary tokens are folded to uppercase in a C or Java program only if the appropriate precompiler option is specified.	G
DESCRIPTOR	BLOB(2MB) NOT NULL	Internal description of the row permission or column mask	S

SYSIBM.SYSCOPY table

The SYSIBM.SYSCOPY table contains information needed for recovery.

Column name	Data type	Description	Use
DBNAME	CHAR(8) NOT NULL	Name of the database.	G
TSNAME	CHAR(8) NOT NULL	Name of the target table space or index space.	G
DSNUM	INTEGER NOT NULL	Data set number within the table space. For partitioned table spaces, this value corresponds to the partition number for a single partition copy, or 0 for a copy of an entire partitioned table space or index space.	G
ICTYPE	CHAR(1) NOT NULL	Type of operation: **A** ALTER **B** REBUILD INDEX **C** CREATE **D** CHECK DATA LOG(NO) (no log records for the range are available for RECOVER utility) **E** RECOVER (to current point) **F** COPY FULL YES **I** COPY FULL NO **J** REORG TABLESPACE or LOAD REPLACE compression dictionary write to log **L** SQL (type of operation) **M** MODIFY RECOVERY utility **P** RECOVER TOCOPY or RECOVER TORBA (partial recovery point) **Q** QUIESCE **R** LOAD REPLACE LOG(YES) **S** LOAD REPLACE LOG(NO) **T** TERM UTILITY command **V** REPAIR VERSIONS utility **W** REORG LOG(NO) **X** REORG LOG(YES) **Y** LOAD LOG(NO) **Z** LOAD LOG(YES)	G
	CHAR(6) NOT NULL	Not used	N

Column name	Data type	Description	Use
START_RBA	CHAR(10) NOT NULL FOR BIT DATA	An 80-bit positive integer that contains the RBA/LRSN of a point in the DB2 recovery log. (The LRSN is the RBA in a data-sharing environment.) • For ICTYPE I or F, the starting point for all updates since the image copy was taken • For ICTYPE J, the RBA/LRSN of the compression dictionary • For ICTYPE M, the RBA of the highest deleted SYSCOPY or SYSLGRNX record • For ICTYPE P, the point after the log-apply phase of point-in-time recovery • For ICTYPE Q, the point after all data sets have been successfully quiesced • For ICTYPE R or S, the end of the log before the start of the LOAD utility and before any data is changed • For ICTYPE T, the end of the log when the utility is terminated • For other values of ICTYPE, the end of the log before the start of the RELOAD phase of the LOAD or REORG utility. A SELECT from SYSIBM.SYSCOPY displays the START_RBA and PIT_RBA columns in either 6-byte or 10-byte format. Before CATENFM of SYSCOPY, the data and the display are in 6-byte format but in all migration modes in utility-output, SYSCOPY columns are displayed in 10-byte format. After CATENFM of SYSCOPY the data and the display are in 10-byte format with non-zero digits in low order 3 bytes. Digits in the low order 3 bytes are unrelated to the conversion of the BSDS or conversion of individual objects to EXTENDED format.	G
FILESEQNO	INTEGER NOT NULL	Tape file sequence number of the copy.	G
DEVTYPE	CHAR(8) NOT NULL	Device type the copy is on.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies. RELCREATED should be used instead.	G
DSNAME	CHAR(44) NOT NULL	For ICTYPE='P' (RECOVER TOCOPY only), "I", or 'F', DSNAME contains the data set name. Otherwise, DSNAME contains the name of the database and table space or index space in the form, *database-name.space-name*, or DSNAME is blank for any row migrated from a DB2 release prior to Version 4.	G
	CHAR(6) NOT NULL	Not used	N

Column name	Data type	Description	Use
SHRLEVEL	CHAR(1) NOT NULL	SHRLEVEL parameter on COPY (for ICTYPE F or I only): **C** Change **R** Reference **blank** Does not describe an image copy or was migrated from Version 1 Release 1 of DB2.	G
DSVOLSER	VARCHAR(1784) NOT NULL	One of the following values: • If the operation is not an image copy operation that creates a FlashCopy® image copy with consistency (an image copy operation with the FLASHCOPY CONSISTENT option), this value is: – A comma-separated list of 6-byte volume serial numbers of the data set, if the data set is not catalogued. – Blank if the data set is cataloged. • If the operation is an image copy operation that creates a FlashCopy image copy with consistency (an image copy operation with the FLASHCOPY CONSISTENT option), this value is a comma-separated list of values of the following form: *memberID-ckptrba* *memberID* is a 3-digit ID for a member of a data sharing group. *ckptrba* is the 12-byte hexadecimal checkpoint RBA for the member.	G
TIMESTAMP	TIMESTAMP NOT NULL WITH DEFAULT	The date and time when the row was inserted. For the COPYTOCOPY utility, this value is the date and time when the row was inserted for the primary local site or primary recovery site copy. For an EXCHANGE DATA statement, this is the time that the statement is run.	G
ICBACKUP	CHAR(2) NOT NULL WITH DEFAULT	Specifies the type of image copy contained in the data set: **blank** LOCALSITE primary copy (first data set named with COPYDDN) **FC** FlashCopy copy **LB** LOCALSITE backup copy (second data set named with COPYDDN) **RP** RECOVERYSITE primary copy (first data set named with RECOVERYDDN) **RB** RECOVERYSITE backup copy (second data set named with RECOVERYDDN)	G
ICUNIT	CHAR(1) NOT NULL WITH DEFAULT	Indicates the media that the image copy data set is stored on: **D** DASD **T** Tape **blank** Medium is neither tape nor DASD, the image copy is from a DB2 release prior to Version 2 Release 3, or ICTYPE is not 'I' or 'F'.	G

Column name	Data type	Description	Use
STYPE	CHAR(1) NOT NULL WITH DEFAULT	When **ICTYPE=A**, the values are: **A** A partition was added to a table. **B** The MEMBER CLUSTER value was changed. **C** A column was added to a table and an index in different commit scopes, or a column was dropped from a table. **D** Either the DSSIZE attribute of the table space was altered or the default value of a column of a table was altered. **E** The data set numbers of a base table and its associated clone table are exchanged. **F** The page size attribute of the table space or index was altered. **G** An index was regenerated **H** The table was altered to hash organization, the size of the hash space was changed, or the hash organization was dropped. The value of the TTYPE column indicates the action taken. **I** The inline length attribute of the LOB column was altered by REORG. **L** The logging attribute of the table space was altered to LOGGED. **M** The MAXPARTITIONS attribute of the table space was altered. **N** An index was altered to not padded **O** The logging attribute of the table space was altered to NOT LOGGED. **P** An index was altered to padded **R** A table was altered to rotate partitions. **S** The SEGSIZE attribute of the table space was altered. **V** A column in a table was altered for a numeric data type change and the column is in an index. **X** A REORG dropped one or more empty partitions from the related table space. **Y** An index was altered to COPY YES **Z** A column that is in the key of an index that was versioned prior to DB2 Version 8 was altered. When **ICTYPE=C**, the values are: **L** The logging attribute of the table space was LOGGED. **O** The logging attribute of the table space was NOT LOGGED. When **ICTYPE=E**, the values are: **B** RECOVER utility with the BACKOUT keyword. **blank** RECOVER utility without the BACKOUT keyword. When **ICTYPE=F**, the values are: **C** DFSMS concurrent copy ("I" instance of the table space) **J** DFSMS concurrent copy ("J" instance of the table space) **N** A FlashCopy copy is not consistent. **Q** Sequential copy is consistent **R** LOAD REPLACE(YES) Values for STYPE continue on next page.	G

Column name	Data type	Description	Use
STYPE (continued)		When **ICTYPE=F**, the values are (continued): **S** LOAD REPLACE(NO) **T** FlashCopy copy is consistent. **U** Sequential copy is not consistent **V** ALTER INDEX NOT PADDED **W** REORG LOG(NO) **X** REORG LOG(YES) **blank** DB2 image copy When **ICTYPE=L**, the value is: **M** Mass DELETE, TRUNCATE TABLE, DROP TABLE, or ALTER TABLE ROTATE PARTITION. The LOWDSNUM column contains the table OBID of the affected table. The MERGECOPY utility, when used to merge an embedded copy with subsequent incremental copies, also produces a record that contains ICTYPE=F and the STYPE of the original image copy (R, S, W, or X). When **ICTYPE = M** and the MODIFY RECOVERY utility was executed to delete SYSCOPY and/or SYSLGRNX records, the value is R. When **ICTYPE=O**, the values are: **B** A table space or partition that was in reordered row format was recovered to a point in time when it was in basic row format. **R** A table space or partition was converted to reordered row format as a result of REORG or LOAD REPLACE. When **ICTYPE=P**, the values are: **B** Recover to a point in time with the BACKOUT YES option was run. **C** Recover to a point in time without using logonly with consistency. **L** Recover to a point in time using logonly without consistency. **M** Recover to a point in time using logonly with consistency. **blank** Recover to a point in time without using logonly without consistency. When **ICTYPE=Q** and option WRITE(YES) is in effect when the quiesce point is taken, the value is W. When **ICTYPE=R or S**, the values are: **A** Resetting REORG pending status **T** First materializing the default value for a row change timestamp column When **ICTYPE=T**, this field indicates which COPY utility was terminated by the TERM UTILITY command or the START DATABASE command with the ACCESS(FORCE) option. The values are: **F** COPY FULL YES **I** COPY FULL NO	

Column name	Data type	Description	Use
STYPE (continued)		When **ICTYPE=W or X**, the values are: **A** Resetting REORG pending status or REBALANCE **T** First materializing the default value for a row change timestamp column For other values of ICTYPE, the value is blank.	
PIT_RBA	CHAR(10) NOT NULL WITH DEFAULT FOR BIT DATA	The meaning of the value depends on the value of the ICTYPE column: **ICTYPE='P'** The LRSN for the point in the DB2 log. (The LRSN is the RBA in a non-data-sharing environment) The value indicates the stop location of a point-in-time recovery. If a record contains ICTYPE='P' and PIT_RBA=X'000000000000', the copy pending status is active and a full image copy is required. If such a record is encountered during fallback processing of RECOVER, the recover job fails, and a point-in-time recovery is required. PIT_RBA can be zero if the point-in-time recovery is completed by the fall-back processing of RECOVER, or if ICTYPE=P from a prior release of DB2. **ICTYPE='F' or 'I' and SHRLEVEL='C'** The current RBA or LRSN that corresponds to the point in the DB2 log when the SHRLEVEL CHANGE copy completes. **ICTYPE=J** The RBA where the compression dictionary is written to the log. In data sharing environments, it is the RBA of the of the member writing to the log. **ICTYPE='M'** The RBA/LRSN for the end of the log when the utility completes. For other all other ICTYPE values, this field contains X'00000000000000000000'. A SELECT from SYSIBM.SYSCOPY displays the START_RBA and PIT_RBA columns in either 6-byte or 10-byte format. Before CATENFM of SYSCOPY, the data and the display are in 6-byte format but in all migration modes in utility-output, SYSCOPY columns are displayed in 10-byte format. After CATENFM of SYSCOPY the data and the display are in 10-byte format with non-zero digits in low order 3 bytes. Digits in the low order 3 bytes are unrelated to the conversion of the BSDS or conversion of individual objects to EXTENDED format.	G
GROUP_MEMBER	CHAR(8) NOT NULL WITH DEFAULT	The DB2 data sharing member name of the DB2 subsystem that performed the operation. This column is blank if the DB2 subsystem was not in a DB2 data sharing environment at the time the operation was performed.	G

Column name	Data type	Description	Use
OTYPE	CHAR(1) NOT NULL WITH DEFAULT 'T'	Type of object that the recovery information is for: **I** Index space **T** Table space	G
LOWDSNUM	INTEGER NOT NULL WITH DEFAULT	Partition number of the lowest partition in the range for SYSCOPY records created for REORG and LOAD REPLACE for resetting a REORG pending status. Version number of an index for SYSCOPY records created for a COPY (ICTYPE=F) of an index space (OTYPE=I). (An index is versioned when a VARCHAR column in the index key is lengthened.) When ICTYPE = F or I, DSNUM = 0 and OTYPE is not equal to I, LOWDSNUM = 1.	G
HIGHDSNUM	INTEGER NOT NULL WITH DEFAULT	Partition number of the highest partition in the range. This column is valid only for SYSCOPY records created for REORG and LOAD REPLACE for resetting REORG pending status.When ICTYPE = F or I, DSNUM = 0 and OTYPE is not equal to I, HIGHDSNUM is the number of the highest partition that is copied.	G
COPYPAGESF	FLOAT(8) NOT NULL WITH DEFAULT -1	Number of pages written to the copy data set. For inline copies, this number might include pages appearing more than once in the copy data set.	G
NPAGESF	FLOAT(8) NOT NULL WITH DEFAULT -1	The number of pages in the table space or index at the time of COPY. This number might include pre-formatted pages that are not actually copied. When **ICTYPE=A, SYTPE=H, and TTYPE=S or D**, this column contains the previous HASHDATAPAGES value. When **ICTYPE=A, SYTPE=H, and TTYPE=A** this column contains zero.	G
CPAGESF	FLOAT(8) NOT NULL WITH DEFAULT -1	Total number of changed pages.	G
JOBNAME	CHAR(8) NOT NULL WITH DEFAULT	Job name of the utility. For changes that cause pending definition changes to object, this column might not be accurate.	G
AUTHID	CHAR(8) NOT NULL WITH DEFAULT	Authorization ID of the utility.For changes that cause pending definition changes to object, this column might not be accurate.	G
OLDEST_VERSION	SMALLINT NOT NULL WITH DEFAULT	When ICTYPE= B, F, I, S, W, or X, the version number of the oldest format of data for an object. For other values of ICTYPE, the value is 0.	G
LOGICAL_PART	INTEGER NOT NULL WITH DEFAULT	Logical partition number.	G

Column name	Data type	Description	Use
LOGGED	CHAR(1) NOT NULL WITH DEFAULT	Indicates the logging attribute of the table space at the time the SYSCOPY record is written: • Y — indicates that the logging attribute of the table space is LOGGED • N — indicates that the logging attribute of the table spaces is NOT LOGGED • blank — indicates that the row was inserted prior to Version 9 or is not specified. For non-LOB table spaces or an index space, blank indicates that the logging attribute is LOGGED.	G

Column name	Data type	Description	Use
TTYPE	CHAR(8) NOT NULL WITH DEFAULT	When **ICTYPE=A and STYPE=B**, this column indicates if the previous value for the MEMBER CLUSTER attribute is being used: **Y** The previous member cluster attribute of the table space is being used. **N** The previous member cluster attribute of the table space is not being used. When **ICTYPE=A and STYPE=C**, this column indicates if a column is added or dropped from a table: **blank** A column was added to a table. **D** A column was dropped from a table. When **ICTYPE=A and STYPE=D**, this column contains the previous DSSIZE attribute value for the table space in units of G, M, or K when the DSSIZE attribute is altered. This column is blank if the default value of a column of a table was altered. When **ICTYPE=A and STYPE=F**, this column indicates the previous page size attribute value for the table space in units of K. When **ICTYPE=A and STYPE=H** this column indicates a change that was applied to the hash organization of the table: **A** Hash organization was added. The record is written when the hash space is materialized at REORG. **D** Hash organization was dropped. The record is written immediately when the ALTER statement is issued. **S** The size of the hash space was changed. The value of the NAPGESF column contains the previous HASHDATAPAGES value. The record is written when the hash space is materialized at REORG. When **ICTYPE=A and STYPE=I**, this column indicates that the inline length of a LOB column was altered: **D** Indicates that REORG decremented the inline length of the LOB column **I** Indicates that REORG incremented the inline length of the LOB column When **ICTYPE=A and STYPE=M**, this column indicates either the previous value of the MAXPARTITIONS attribute for the table space or the type of table space conversion that was performed on the table space. **I** The table space was converted from a single-table simple table space to a partition-by-growth universal table space. **n** The previous value of the MAXPARTITIONS attribute for the table space. **S** The table space was converted from single-table segmented table space to a partition-by-growth universal table space.	G

Column name	Data type	Description	Use
TTYPE (cont)		When **ICTYPE=A and STYPE=S**, this column indicates either the previous value of the SEGSIZE attribute for the table space or the type of table space conversion that was performed on the table space. **n** The previous value of the SEGSIZE attribute for the table space. **P** The table space was converted from a partitioned table space to a range-partitioned universal table space. When **ICTYPE=E**, this column indicates if the full recovery reset the object: **blank** The full recovery reset the object **N** The full recovery did not reset the object When **ICTYPE=F and STYPE=N, Q, T, or U**, this column indicates the utility that made the FlashCopy: **A** LOAD RESUME LOG NO **B** REBUILD **C** COPY **D** LOAD RESUME LOG YES **E** LOAD SHRLEVEL CHANGE **L** LOAD **P** REPAIR **R** LOAD REPLACE LOG YES **S** LOAD REPLACE LOG NO **T** COPYTOCOPY **W** REORG TABLESPACE LOG NO **X** REORG TABLESPACE LOG YES When **ICTYPE=I**, TTYPE of S indicates that the directory pages for the index image copy are at the front of each partition and are indicated with a 'V' or '8'. When **ICTYPE=P, R, S, W, X**, this column provides additional diagnostic information: **B** Indicates that the RBA or LRSN format changed to basic 6-byte format. **BRF** Indicates that the row format is the basic row format. **BRF I** Indicates that the row format is the basic row format, and the FORMAT INTERNAL option was specified. **E** Indicates that the RBA or LRSN format changed to extended 10-byte format. **F** Indicates that the REORG utility was run with the FASTSWITCH YES option. **RRF** Indicates that the row format is the reordered row format. **RRF I** Indicates that the row format is the reordered row format, and the FORMAT INTERNAL option was specified. **S** Indicates that the REORG utility was run with the FASTSWITCH NO option.	

Column name	Data type	Description	Use
TTYPE (cont)		When **ICTYPE=M and STYPE=R**, this column indicates whether the MODIFY RECOVERY utility deleted rows from SYSIBM.SYSLGRNX. **blank** MODIFY RECOVERY deleted rows from SYSIBM.SYSLGRNX. **N** MODIFY RECOVERY did not delete rows from SYSIBM.SYSLGRNX. When **ICTYPE=T**, TTYPE of B indicates that a broken page was detected during copy. When **ICTYPE=W or X and STYPE=H**, this column indicates the prior value of HASHDATAPAGES. When **ICTYPE=Y or Z**, this column indicates whether the object was loaded when the FORMAT INTERNAL option was specified. **blank** Indicates that the FORMAT INTERNAL option was not specified during LOAD. **I** Indicates that the FORMAT INTERNAL option was specified during LOAD. When **ICTYPE=A-A, A-R, B, C, P, R, S, W, or X** and the page format was changed by the ALTER ADD PARTITION, ALTER ROTATE PARTITION, CREATE, LOAD REPLACE, REBUILD, REORG, or RECOVER utilities: **B** Indicates that the page format was converted to basic page format with 6-byte RBA or LRSN values. **E** Indicates that the page format was converted to extended page format with 10-byte RBA or LRSN values. When **ICTYPE=A and STYPE=A or R:** **B** Indicates that the page format was converted to basic page format with 6-byte RBA or LRSN values. **E** Indicates that the page format was converted to extended page format with 10-byte RBA or LRSN values.	
INSTANCE	SMALLINT NOT NULL WITH DEFAULT 1	When STYPE = E and ICTYPE = A, INSTANCE indicates the data set instance number of a base object after an EXCHANGE statement completes. The value of the INSTANCE column for the last data exchange will match the value of the INSTANCE column for the SYSIBM.SYSTABLESPACE table. For an image copy, INSTANCE indicates the instance number of the current base objects (table and index).	G
RELCREATED	CHAR(1) NOT NULL WITH DEFAULT	The release of DB2 that is used to create the object. Blank if created prior to Version 9. See Release dependency indicators for all other values.	G

Column name	Data type	Description	Use
MODECREATED	CHAR(2) NOT NULL WITH DEFAULT	The latest mode to which the DB2 subsystem had been migrated when the SYSCOPY record was written: **C** Conversion mode **E** Enabling-new-function mode **N** New-function mode	

SYSIBM.SYSCTXTTRUSTATTRS table

The SYSIBM.SYSCTXTTRUSTATTRS table contains one row for each list of attributes for a given trusted context.

Column name	Data type	Description	Use
CONTEXTID	INTEGER NOT NULL	The internal trusted context ID.	G
NAME	VARCHAR(128) NOT NULL	Name of the trust attribute. Possible values including the following attributes: • An IPv4 address is represented as a dotted decimal IP address. An example of an IPv4 address is '9.112.46.111'. • An IPv6 address is represented as a colon hexadecimal address. An example of an IPv6 address is '2001:0DB8:0000:0000:0008:0800:200C:417A', which can also be expressed in a compressed form as '2001:DB8::8:800:200C:417A'. • A domain name which is converted to an IP address by the domain name server where a resulting IPv4 or IPv6 address is determined. • A job or started task name for local applications. If the job name ends with *, any job name that matches the characters prior to * in the specified job name are considered for establishing the trusted connection. • A network access security zone name in the RACF SERVAUTH class.	G
VALUE	VARCHAR(254) NOT NULL	The value of the trust attribute.	G
CREATEDTS	TIMESTAMP NOT NULL	The time when the attribute is created.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

SYSIBM.SYSDATABASE table

The SYSIBM.SYSDATABASE table contains one row for each database, except for database DSNDB01.

Column name	Data type	Description	Use
NAME	VARCHAR(24) NOT NULL	Database name.	G
CREATOR	VARCHAR(128) NOT NULL	Authorization ID of the owner of the database.	G
STGROUP	VARCHAR(128) NOT NULL	Name of the default storage group of the database; blank for a system database.	G
BPOOL	CHAR(8) NOT NULL	Name of the default buffer pool of the table space; blank for a system table space.	G
DBID	SMALLINT NOT NULL	Internal identifier of the database. If there were 32511 databases or more when this database was created, the DBID is a negative number.	S
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies. RELCREATED should be used instead.	G
CREATEDBY	VARCHAR(128) NOT NULL WITH DEFAULT	Primary authorization ID of the user who created the database.	G
	CHAR(1) NOT NULL WITH DEFAULT	Not used	N
	TIMESTAMP NOT NULL WITH DEFAULT	Not used	N
TYPE	CHAR(1) NOT NULL WITH DEFAULT	Type of database: **blank** Not a work file database or a TEMP database. **T** A TEMP file database. **W** A work file database. The database is DSNDB07, or it was created with the WORKFILE clause and used as a work file database by a member of a DB2 data sharing group.	G
GROUP_MEMBER	VARCHAR(24) NOT NULL WITH DEFAULT	The DB2 data sharing member name of the DB2 subsystem that uses this work file database. This column is blank if the work file database was not created in a DB2 data sharing environment, or if the database is not a work file database as indicated by the TYPE column.	G
CREATEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the CREATE statement was executed for the database. For DSNDB04 and DSNDB06, the value is '1985-04-01.00.00.00.000000'.	G

Column name	Data type	Description	Use
ALTEREDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the most recent ALTER DATABASE statement was applied. If no ALTER DATABASE statement has been applied, ALTEREDTS has the value of CREATEDTS.	G
ENCODING_SCHEME	CHAR(1) NOT NULL WITH DEFAULT 'E'	Default encoding scheme for the database: **E** EBCDIC **A** ASCII **U** UNICODE **blank** For DSNDB04, a work file database, and a TEMP database.	G
SBCS_CCSID	INTEGER NOT NULL WITH DEFAULT	Default SBCS CCSID for the database. For a TEMP database, a work file database, or a database created in a DB2 release prior to Version 5, the value is 0.	G
DBCS_CCSID	INTEGER NOT NULL WITH DEFAULT	Default DBCS CCSID for the database. If mixed data is not used and the CCSID for the database is defined as EBCDIC or ASCII, the default value is 0. For a TEMP database, a work file database, or a database created in a DB2 release prior to Version 5, the value is 0.	G
MIXED_CCSID	INTEGER NOT NULL WITH DEFAULT	Default mixed CCSID for the database. If mixed data is not used and the CCSID for the database is defined as EBCDIC or ASCII, the default value is 0. For a TEMP database, a work file database, or a database created in a DB2 release prior to Version 5, the value is 0.	G
INDEXBP	CHAR(8) NOT NULL WITH DEFAULT 'BP0'	Name of the default buffer pool for indexes.	G
IMPLICIT	CHAR(1) NOT NULL WITH DEFAULT 'N'	Indicates whether the database was implicitly created: **Y** The database was implicitly created **N** The database was explicitly created	G
CREATORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of creator: **blank** Authorization ID **L** Role	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. See Release dependency indicators for the values.	G

SYSIBM.SYSDATATYPES table

The SYSIBM.SYSDATATYPES table contains one row for each user-defined type defined to the system.

Column name	Data type	Description	Use
SCHEMA	VARCHAR(128) NOT NULL	Schema of the data type.	G
OWNER	VARCHAR(128) NOT NULL	Owner of the data type.	G
NAME	VARCHAR(128) NOT NULL	Name of the data type.	G
CREATEDBY	VARCHAR(128) NOT NULL	Primary authorization ID of the user who created the data type.	G
SOURCESCHEMA	VARCHAR(128) NOT NULL	Schema of the source data type.	G
SOURCETYPE	VARCHAR(128) NOT NULL	Name of the source type.	G
METATYPE	CHAR(1) NOT NULL	The class of data type: **A** User-defined ordinary array type **L** User-defined associative array type **T** Distinct type	G
DATATYPEID	INTEGER NOT NULL	Internal identifier of the data type.	S
SOURCETYPEID	INTEGER NOT NULL	Internal ID of the built-in data type on which the distinct type or array elements are based.	S
LENGTH	INTEGER NOT NULL	Maximum length or precision for a data type that is based on the IBM-defined DECIMAL data type. The data type can be a distinct type or an array type.	G
SCALE	SMALLINT NOT NULL	One of the following values: • For a data type that is based on the IBM-defined DECIMAL data type, the scale. The data type can be a distinct type or an array type. Number of fractional second digits for a data type that is based on the IBM-defined timestamp or timestamp with time zone type. • For a data type that is based on the IBM-defined TIMESTAMP or TIMESTAMP WITH TIME ZONE type, the number of fractional-second digits. • For any other data type, the value is 0. If the value is a timestamp, the LENGTH is 10 and the SCALE is 0, the number of fractional second digits is 6.	G

Column name	Data type	Description	Use
SUBTYPE	CHAR(1) NOT NULL	Subtype of the data type, if the source type is one of the character types. The data type can be a distinct type or an array type. Possible values are: **B** The subtype is FOR BIT DATA. **S** The subtype is FOR SBCS DATA. **M** The subtype is FOR MIXED DATA. **blank** The source type is not a character type.	G
CREATEDTS	TIMESTAMP NOT NULL	Time when the data type was created.	G
ENCODING_SCHEME	CHAR(1) NOT NULL	Encoding scheme of the data type: **A** ASCII **E** EBCDIC **U** UNICODE	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies. RELCREATED should be used instead.	G
REMARKS	VARCHAR(762) NOT NULL	A character string provided by the user with the COMMENT statement.	G
OWNERTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of owner: **blank** Authorization ID **L** Role	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. See Release dependency indicators for the values.	G
INLINE_LENGTH	INTEGER NOT NULL WITH DEFAULT -1	The inline length attribute of the type if it is based on a LOB source type: **-1** This type does not specify INLINE LENGTH **greater than or equal to 0** The inline length attribute (in byte) of the type if it is based on a LOB source type	G
ARRAYLENGTH	BIGINT NOT NULL WITH DEFAULT	Maximum cardinality, if the data type is an array type. For all other data types, the value is 0.	G
ARRAYINDEXTYPEID	INTEGER NOT NULL WITH DEFAULT	Data type of the index, if the data types is an associative array type. For all other data types, the value is 0.	G
ARRAYINDEX- TYPELEN	BIGINT NOT NULL WITH DEFAULT	Maximum length of the array index, if the data types is an associative array type. For all other data types, the value is 0.	G
ARRAYINDEX- SUBTYPE	CHAR(1) NOT NULL WITH DEFAULT	Subtype of the array index: **B** The subtype is FOR BIT DATA. **S** The subtype is FOR SBCS DATA. **M** The subtype is FOR MIXED DATA. **blank** The array index is not a character type.	G

SYSIBM.SYSDBAUTH table

The SYSIBM.SYSDBAUTH table records the privileges that are held by users over databases.

Column name	Data type	Description	Use
GRANTOR	VARCHAR(128) NOT NULL	Authorization ID or role of the user who granted the privileges. Could also be PUBLIC.	G
GRANTEE	VARCHAR(128) NOT NULL	Application ID of the user who holds the privilege. Could also be PUBLIC for a grant to PUBLIC.	G
NAME	VARCHAR(24) NOT NULL	Database name.	G
	CHAR(12) NOT NULL	Internal use only	I
	CHAR(6) NOT NULL	Not used	N
	CHAR(8) NOT NULL	Not used	N
GRANTEETYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of grantee: **blank** Authorization ID **L** Role	G
	CHAR(1) NOT NULL	Not used	N
AUTHHOWGOT	CHAR(1) NOT NULL	Authorization level of the user from whom the privileges were received. This authorization level is not necessarily the highest authorization level of the grantor. **blank** Not applicable **C** DBCTRL **D** DBADM **E** SECADM **G** ACCESSCTRL **L** SYSCTRL **M** DBMAINT **S** SYSADM	G
CREATETABAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can create tables within the database: **blank** Privilege is not held **G** Privilege held with the GRANT option **Y** Privilege is held without the GRANT option	G
CREATETSAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can create table spaces within the database: **blank** Privilege is not held **G** Privilege held with the GRANT option **Y** Privilege is held without the GRANT option	G

Column name	Data type	Description	Use
DBADMAUTH	CHAR(1) NOT NULL	Whether the GRANTEE has DBADM authority over the database: **blank** Privilege is not held **G** Privilege held with the GRANT option **Y** Privilege is held without the GRANT option	G
DBCTRLAUTH	CHAR(1) NOT NULL	Whether the GRANTEE has DBCTRL authority over the database: **blank** Privilege is not held **G** Privilege held with the GRANT option **Y** Privilege is held without the GRANT option	G
DBMAINTAUTH	CHAR(1) NOT NULL	Whether the GRANTEE has DBMAINT authority over the database: **blank** Privilege is not held **G** Privilege held with the GRANT option **Y** Privilege is held without the GRANT option	G
DISPLAYDBAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can issue the DISPLAY command for the database: **blank** Privilege is not held **G** Privilege held with the GRANT option **Y** Privilege is held without the GRANT option	G
DROPAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can issue the ALTER DATABASE and DROP DATABASE statement: **blank** Privilege is not held **G** Privilege held with the GRANT option **Y** Privilege is held without the GRANT option	G
IMAGCOPYAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can use the COPY, MERGECOPY, MODIFY, and QUIESCE utilities on the database: **blank** Privilege is not held **G** Privilege held with the GRANT option **Y** Privilege is held without the GRANT option	G
LOADAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can use the LOAD utility to load tables in the database: **blank** Privilege is not held **G** Privilege held with the GRANT option **Y** Privilege is held without the GRANT option	G
REORGAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can use the REORG utility to reorganize table spaces and indexes in the database: **blank** Privilege is not held **G** Privilege held with the GRANT option **Y** Privilege is held without the GRANT option	G
RECOVERDBAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can use the RECOVER and REPORT utilities on table spaces in the database: **blank** Privilege is not held **G** Privilege held with the GRANT option **Y** Privilege is held without the GRANT option	G
REPAIRAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can use the DIAGNOSE and REPAIR utilities on table spaces and indexes in the database: **blank** Privilege is not held **G** Privilege held with the GRANT option **Y** Privilege is held without the GRANT option	G

Column name	Data type	Description	Use
STARTDBAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can use the START command against the database: **blank** Privilege is not held **G** Privilege held with the GRANT option **Y** Privilege is held without the GRANT option	G
STATSAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can use the CHECK and RUNSTATS utilities against the database: **blank** Privilege is not held **G** Privilege held with the GRANT option **Y** Privilege is held without the GRANT option	G
STOPAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can issue the STOP command against the database: **blank** Privilege is not held **G** Privilege held with the GRANT option **Y** Privilege is held without the GRANT option	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
GRANTEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the GRANT statement was executed.	G
GRANTORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of grantor: **blank** Authorization ID **L** Role	G

SYSIBM.SYSDBRM table

The SYSIBM.SYSDBRM table contains one row for each DBRM of each application plan.

Column name	Data type	Description	Use
NAME	VARCHAR(24) NOT NULL	Name of the DBRM.	G
TIMESTAMP	CHAR(8) NOT NULL FOR BIT DATA	Consistency token.	S
PDSNAME	VARCHAR(132) NOT NULL	Name of the partitioned data set of which the DBRM is a member.	G
PLNAME	VARCHAR(24) NOT NULL	Name of the application plan of which this DBRM is a part.	G
PLCREATOR	VARCHAR(128) NOT NULL	Authorization ID of the owner of the application plan.	G
	CHAR(8) NOT NULL	Not used	N
	CHAR(6) NOT NULL	Not used	N
QUOTE	CHAR(1) NOT NULL	SQL string delimiter for the SQL statements in the DBRM: **N** Apostrophe **Y** Quotation mark	G
COMMA	CHAR(1) NOT NULL	Decimal point representation for SQL statements in the DBRM: **N** Period **Y** Comma	G
HOSTLANG	CHAR(1) NOT NULL	The host language used: **B** Assembler language **C** OS/VS COBOL **D** C **F** Fortran **P** PL/I **2** VS COBOL II or IBM COBOL Release 1 (formerly called COBOL/370) **3** IBM COBOL (Release 2 or subsequent releases) **4** C++	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies. RELCREATED should be used instead.	G
CHARSET	CHAR(1) NOT NULL WITH DEFAULT	Indicates whether the system CCSID for SBCS data was 290 (Katakana) when the program was precompiled: **A** No **K** Yes	G

Column name	Data type	Description	Use
MIXED	CHAR(1) NOT NULL WITH DEFAULT	Indicates if mixed data was in effect when the application program was precompiled (for more on when mixed data is in effect, see "Character strings" on page 84): **N** No **Y** Yes	G
DEC31	CHAR(1) NOT NULL WITH DEFAULT	Indicates whether DEC31 was in effect when the program was precompiled (for more on when DEC31 is in effect, see "Arithmetic with two decimal operands" on page 244): **blank** No **Y** Yes	G
VERSION	VARCHAR(122) NOT NULL WITH DEFAULT	Version identifier for the DBRM.	G
PRECOMPTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the DBRM was precompiled.	G
PLCREATORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of creator: **blank** Authorization ID **L** Role	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. See Release dependency indicators for the values.	G

SYSIBM.SYSDEPENDENCIES table

The SYSIBM.SYSDEPENDENCIES table records the dependencies between objects.

Column name	Data type	Description	Use
BNAME	VARCHAR(128) NOT NULL	Name of the object on which another object is dependent. If BTYPE is F, the name is the specific name of the function. If BTYPE is W or Z, the name is the name of the table for which the period is defined.	G
BSCHEMA	VARCHAR(128) NOT NULL	Schema or qualifier of the object on which another object is dependent.	G
BCOLNAME	VARCHAR(128) NOT NULL WITH DEFAULT	Column name of the object on which another object is dependent.	G
BCOLNO	SMALLINT NOT NULL WITH DEFAULT	Column number of the object on which another object is dependent.	G
BTYPE	CHAR(1) NOT NULL	The type of object that is identified by BNAME, BSCHEMA, and BCOLNAME: **C** Column **E** INSTEAD OF trigger **F** Function **G** Global temporary table **I** Index **M** Materialized query table **O** Procedure **P** Partitioned table space **Q** Sequence **R** Table space **S** Synonym **T** Table **U** Distinct type **V** View **W** SYSTEM_TIME period **Z** BUSINESS_TIME period **0 (zero)** Alias	G
BOWNER	VARCHAR(128) NOT NULL WITH DEFAULT	Authorization ID of the owner of the object on which another object is dependent.	G

Column name	Data type	Description	Use
BOWNERTYPE	CHAR(1) NOT NULL	Type of creator of the object on which another object is dependent: **L** Role **blank** Authorization ID that is not a role	G
DNAME	VARCHAR(128) NOT NULL	Name of the object that has dependencies on another object.	G
DSCHEMA	VARCHAR(128) NOT NULL	Schema or qualifier of the object that has dependencies on another object.	G
DVERSION	VARCHAR(122) NOT NULL WITH DEFAULT	The version identifier of the object that is identified by DSCHEMA and DNAME if the object has a version. This column contains a zero length string for the objects that are created prior to Version 10 and for the rows that correspond to objects without versions.	G
DCOLNAME	VARCHAR(128) NOT NULL	Column name of the object that has dependencies on another object.	G
DCOLNO	SMALLINT NOT NULL WITH DEFAULT	Column number of the object that has dependencies on another object.	G
DTYPE	CHAR(1) NOT NULL	The type of the object that is identified by DSCHEMA, DNAME, DCOLNAME, and DVERSION: **B** Trigger **C** Generated column **F** Function **I** Index **M** Materialized query table **O** Procedure **V** View **X** Row permission **Y** Column mask	G
DOWNER	VARCHAR(128) NOT NULL	Authorization ID of the owner of the object that has dependencies on another object.	G
DOWNERTYPE	CHAR(1) NOT NULL	Type of creator of the object that has dependencies on another object: **L** Role **blank** Authorization ID if not a role	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

Column name	Data type	Description	Use
BAUTH	SMALLINT NOT NULL WITH DEFAULT	The privilege that is held on the object on which another object is dependent.	G

SYSIBM.SYSDUMMY1 table

The SYSIBM.SYSDUMMY1 table contains one row. The table is used for SQL statements in which a table reference is required, but the contents of the table are not important.

Unlike the other catalog tables, which reside in Unicode table spaces, SYSIBM.SYSDUMMY1 resides in table space SYSEBCDC, which is an EBCDIC table space.

Column name	Data type	Description	Use
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

SYSIBM.SYSDUMMYA table

The SYSIBM.SYSDUMMYA table contains one row. The table is used for SQL statements in which a table reference is required, but the contents of the table are not important.

SYSIBM.SYSDUMMYA resides in table space SYSTSASC, which is a ASCII table space.

Column name	Data type	Description	Use
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

SYSIBM.SYSDUMMYE table

The SYSIBM.SYSDUMMYE table contains one row. The table is used for SQL statements in which a table reference is required, but the contents of the table are not important.

SYSIBM.SYSDUMMYE resides in table space SYSEBCDC, which is a EBCDIC table space.

Column name	Data type	Description	Use
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

SYSIBM.SYSDUMMYU table

The SYSIBM.SYSDUMMYU table contains one row. The table is used for SQL statements in which a table reference is required, but the contents of the table are not important.

SYSIBM.SYSDUMMYU resides in table space SYSTSUNI, which is a UNICODE table space.

Column name	Data type	Description	Use
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

SYSIBM.SYSENVIRONMENT table

The SYSIBM.SYSENVIRONMENT table records the environment variables when an object is created.

Column name	Data type	Description	Use
ENVID	INTEGER NOT NULL	Internal identifier of the environment.	G
CURRENT_SCHEMA	VARCHAR(128) NOT NULL	The current schema.	G
RELCREATED	CHAR(1) NOT NULL	The release when the environment information is created. SeeRelease dependency indicators for values.	G
PATHSCHEMAS	VARCHAR(2048) NOT NULL	The schema path.	G
APPLICATION_ ENCODING_CCSID	INTEGER NOT NULL	The CCSID of the application environment.	G
ORIGINAL_ ENCODING_CCSID	INTEGER NOT NULL	The original CCSID of the statement text string.	G
DECIMAL_POINT	CHAR(1) NOT NULL	The decimal point indicator: **C** Comma **P** Period	G
MIN_DIVIDE_SCALE	CHAR(1) NOT NULL	The minimum divide scale: **N** The usual rules apply for decimal division in SQL **Y** Retain at lease three digits to the right of the decimal point after any decimal division.	G
STRING_DELIMITER	CHAR(1) NOT NULL	The string delimiter that is used in COBOL string constants: **A** Apostrophe (') **Q** Quote (")	G
SQL_STRING_ DELIMITER	CHAR(1) NOT NULL	The SQL string delimiter that is used in string constants: **A** Apostrophe (') **Q** Quote (")	G
MIXED_DATA	CHAR(1) NOT NULL	Uses mixed DBCS data: **N** No mixed data **Y** Mixed data	G
DECIMAL_ ARITHMETIC	CHAR(1) NOT NULL	The rules that are to be used for CURRENT PRECISION and when both operands in a decimal operation have a precision of 15 or less: **1** DEC15 specifies that the rules do not allow a precision greater than 15 digits **2** DEC31 specifies that the rules allow a precision of up to 31 digits	G

Column name	Data type	Description	Use
DATE_FORMAT	CHAR(1) NOT NULL	The date format: **I** ISO - yyyy-mm-dd **J** JIS - yyyy-mm-dd **U** USA - mm/dd/yyyy **E** EUR - dd.mm.yyyy **L** Locally defined by an installation exit routine	G
TIME_FORMAT	CHAR(1) NOT NULL	The time format: **I** ISO - hh.mm.ss **J** JIS - hh.mm.ss **U** USA - hh:mm AM or hh:mm PM **E** EUR - hh.mm.ss **L** Locally defined by an installation exit routine	G
FLOAT_FORMAT	CHAR(1) NOT NULL	The floating point format: **I** IEEE floating point format **S** System/390 floating point format	G
HOST_LANGUAGE	CHAR(8) NOT NULL	The host language: • ASM • C • CPP • IBMCOB • PLI • FORTRAN	G
CHARSET	CHAR(1) NOT NULL	The character set: **A** Alphanumeric	G
FOLD	CHAR(1) NOT NULL	FOLD is only applicable when HOST_LANGUAGE is C or CPP. Otherwise FOLD is blank. **N** Lower case letters in SBCS ordinary identifiers are not folded to uppercase **Y** Lower case letters in SBCS ordinary identifiers are folded to uppercase **blank** Not applicable	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
ROUNDING	CHAR(1) NOT NULL WITH DEFAULT	The rounding mode that is used when arithmetic and casting operations are performed on DECFLOAT data: **C** ROUND_CEILING **D** ROUND_DOWN **F** ROUND_FLOOR **G** ROUND_HALF_DOWN **E** ROUND_HALF_EVEN **H** ROUND_HALF_UP **U** ROUND_UP	G

SYSIBM.SYSFIELDS table

The SYSIBM.SYSFIELDS table contains one row for every column that has a field procedure.

Column name	Data type	Description	Use
TBCREATOR	VARCHAR(128) NOT NULL	Schema or qualifier of the table that contains the column.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table that contains the column.	G
COLNO	SMALLINT NOT NULL	Numeric place of this column in the table.	G
NAME	VARCHAR(128) NOT NULL	Name of the column.	G
FLDTYPE	VARCHAR(24) NOT NULL	Data type of the encoded values in the field (This columns might contain statistical values from a prior release.): **INTEGER** Large integer **SMALLINT** Small integer **FLOAT** Floating-point **CHAR** Fixed-length character string **VARCHAR** Varying-length character string **DECIMAL** Decimal **GRAPHIC** Fixed-length graphic string **VARG** Varying-length graphic string	G
LENGTH	SMALLINT NOT NULL	The length attribute of the field; or, for a decimal field, its precision.(This columns might contain statistical values from a prior release.) The number does not include the internal prefixes that can be used to record actual length and null state. **INTEGER** 4 **SMALLINT** 2 **FLOAT** 8 **CHAR** Length of string **VARCHAR** Maximum length of string **DECIMAL** Precision of number **GRAPHIC** Number of DBCS characters **VARG** Maximum number of DBCS characters	G
SCALE	SMALLINT NOT NULL	Scale if FLDTYPE is DECIMAL; otherwise, the value is 0.	G

Column name	Data type	Description	Use
FLDPROC	VARCHAR(24) NOT NULL	For a row describing a field procedure, the name of the procedure. (This columns might contain statistical values from a prior release.)	G
WORKAREA	SMALLINT NOT NULL	For a row describing a field procedure, the size, in bytes, of the work area required for the encoding and decoding of the procedure. (This columns might contain statistical values from a prior release.)	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
EXITPARML	SMALLINT NOT NULL	For a row describing a field procedure, the length of the field procedure parameter value block. (This columns might contain statistical values from a prior release.)	G
PARMLIST	VARCHAR(735) NOT NULL	For a row describing a field procedure, the parameter list following FIELDPROC in the statement that created the column, with insignificant blanks removed. (This columns might contain statistical values from a prior release.)	G
EXITPARM	VARCHAR(1530) NOT NULL FOR BIT DATA	For a row describing a field procedure, the parameter value block of the field procedure (the control block passed to the field procedure when it is invoked). (This columns might contain statistical values from a prior release.)	G

SYSIBM.SYSFOREIGNKEYS table

The SYSIBM.SYSFOREIGNKEYS table contains one row for every column of every foreign key.

Column name	Data type	Description	Use
CREATOR	VARCHAR(128) NOT NULL	Schema or qualifier of the table that contains the column.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table that contains the column.	G
RELNAME	VARCHAR(128) NOT NULL	Constraint name for the constraint for which the column is part of the foreign key.	G
COLNAME	VARCHAR(128) NOT NULL	Name of the column.	G
COLNO	SMALLINT NOT NULL	Numeric place of the column in its table.	G
COLSEQ	SMALLINT NOT NULL	Numeric place of the column in the foreign key.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

SYSIBM.SYSINDEXCLEANUP table

The rows in the SYSIBM.SYSINDEXCLEANUP table specify time windows to control index cleanup processing. Each row specifies a time window to enable or disable the cleanup of pseudo-deleted index entries for specific database objects.

Column name	Data type	Description	Use
DBNAME	VARCHAR(24)	The name of the database that contains the index space.	G
INDEXSPACE	VARCHAR(24)	The name of the index space.	G
ENABLE_DISABLE	CHAR(1) NOT NULL	Specifies whether the row enables or disables cleanup for the specified index space. **'E'** Enabled **'D'** Disabled	G
MONTH_WEEK	CHAR(1) NOT NULL	Indicates the meaning of the value of the DAY column: **'M'** The value indicates the day of the month. **'W'** The value indicates a day of the week.	G
MONTH	SMALLINT	The month in which the time window applies. For example a 1 value indicates January and a 12 value indicates December. If this column contains NULL, the time window applies to all months. If the value of the MONTH_WEEK column is 'W', this value must be NULL.	G
DAY	SMALLINT	The day of the month or the day of the week for which the time window applies, as specified by the value of the MONTH_WEEK column. For example, if MONTH_WEEK='W', a 1 value indicates Monday and 7 indicates Sunday. If the value of this column is NULL, the time window applies to every day of the month or every day of the week.	G
START_TIME	TIME	The local time at the beginning of the time window specified by the row. When this column contains a null value, the row applies at all times on the specified days. This column must contain NULL if the END_TIME column contains NULL.	G
END_TIME	TIME	The local time at the end of the time window specified by the row. When this column contains a null value, the row applies at all times on the specified days. This column must contain NULL if the START_TIME column contains NULL.	G

Related tasks:

Controlling index cleanup processing (DB2 Performance)

Related reference:

INDEX CLEANUP THREADS field (INDEX_CLEANUP_THREADS subsystem parameter) (DB2 Installation and Migration)

SYSIBM.SYSINDEXES table

The SYSIBM.SYSINDEXES table contains one row for every index.

Column name	Data type	Description	Use
NAME	VARCHAR(128) NOT NULL	Name of the index.	G
CREATOR	VARCHAR(128) NOT NULL	The schema of the index.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table on which the index is defined.	G
TBCREATOR	VARCHAR(128) NOT NULL	The schema of the table.	G
UNIQUERULE	CHAR(1) NOT NULL	Whether the index is unique: **C** Yes, and it is used to enforce the uniqueness of a UNIQUE constraint or hash key columns. **D** No (duplicates are allowed) **U** Yes **P** Yes, and it is a primary index (As in prior releases of DB2, a value of P is used for primary keys that are used to enforce a referential constraint.) **N** Yes, and it is defined with UNIQUE WHERE NOT NULL **R** Yes, and it is an index used to enforce the uniqueness of a non-primary parent key **G** Yes, and it is an index used to enforce the uniqueness of values in a column defined as ROWID GENERATED BY DEFAULT **X** Yes, and it is an index used to enforce the uniqueness of values in a column that is used to identify or find XML values associated with a specific row.	G
COLCOUNT	SMALLINT NOT NULL	The number of columns in the key.	G
CLUSTERING	CHAR(1) NOT NULL	Whether CLUSTER was specified for the index: **N** No **Y** Yes	G
CLUSTERED	CHAR(1) NOT NULL	Whether the table is actually clustered by the index: **N** A significant number of rows are not in clustering order, or statistics have not been gathered. **Y** Most of the rows are in clustering order. **blank** Not applicable. This is an updatable column that can also be changed by the RUNSTATS utility. For a sparse index, the statistic is based on the actual contents of the index.	G

Column name	Data type	Description	Use
DBID	SMALLINT NOT NULL	Internal identifier of the database.	S
OBID	SMALLINT NOT NULL	Internal identifier of the index fan set descriptor.	S
ISOBID	SMALLINT NOT NULL	Internal identifier of the index page set descriptor.	S
DBNAME	VARCHAR(24) NOT NULL	Name of the database that contains the index.	G
INDEXSPACE	VARCHAR(24) NOT NULL	Name of the index space.	G
FIRSTKEYCARD	INTEGER NOT NULL	Not used	N
FULLKEYCARD	INTEGER NOT NULL	Not used	N
NLEAF	INTEGER NOT NULL	Number of active leaf pages in the index. The value is -1 if statistics have not been gathered. This is an updatable column.	S
NLEVELS	SMALLINT NOT NULL	Number of levels in the index tree. If the index is partitioned, it is the maximum of the number of levels in the index tree for all the partitions. The value is -1 if statistics have not been gathered. This is an updatable column.	S
BPOOL	CHAR(8) NOT NULL	Name of the buffer pool used for the index.	G
PGSIZE	SMALLINT NOT NULL	Contains the value 4, 8, 16, or 32 which indicates the size, in KB, of the leaf pages in the index. If the index was created prior to Version 9, the value will be 4096 for a 4 KB page size.	G
ERASERULE	CHAR(1) NOT NULL	Whether the data sets are erased when dropped. The value is meaningless if the index is partitioned: **N** No **Y** Yes	G
	VARCHAR(24) NOT NULL	Not used	N
CLOSERULE	CHAR(1) NOT NULL	Whether the data sets are candidates for closure when the limit on the number of open data sets is reached: **N** No **Y** Yes	G

Column name	Data type	Description	Use
SPACE	INTEGER NOT NULL	Number of kilobytes of DASD storage allocated to the index, as determined by the last execution of the STOSPACE utility. The value is 0 if the index is not related to a storage group, or if STOSPACE has not been run. If the index space is partitioned, the value is the total kilobytes of DASD storage allocated to all partitions that are defined in a storage group.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies. RELCREATED should be used instead.	G
CLUSTERRATIO	SMALLINT NOT NULL WITH DEFAULT	Percentage of rows that are in clustering order. For a partitioning index, it is the weighted average of all index partitions in terms of the number of rows in the partition. The value is 0 if statistics have not been gathered. The value is -2 if the index is for an auxiliary table. This is an updatable column. For a sparse index, the statistic is based on the actual contents of the index.	S
CREATEDBY	VARCHAR(128) NOT NULL WITH DEFAULT	Primary authorization ID of the user who created the index.	G
	SMALLINT NOT NULL	Internal use only	I
	SMALLINT NOT NULL	Not used	N
STATSTIME	TIMESTAMP NOT NULL WITH DEFAULT	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.00.000000'. This is an updatable column.	G
INDEXTYPE	CHAR(1) NOT NULL WITH DEFAULT	The index type: **2** Type 2 index or a hash overflow index on non-partitioned tables. **blank** Type 1 index **D** Data-partitioned secondary index **P** An index that is both partitioned and is a partitioning index (index that is on a table that uses table-controlled partitioning).	G
FIRSTKEYCARDF	FLOAT NOT NULL WITH DEFAULT -1	Number of distinct values of the first key column. This number is an estimate if updated while collecting statistics on a single partition. The value is -1 if statistics have not been gathered. This is an updatable column. For a sparse index, the statistic is based on the actual contents of the index.	S

Column name	Data type	Description	Use
FULLKEYCARDF	FLOAT NOT NULL WITH DEFAULT -1	Number of distinct values of the key. The value is -1 if statistics have not been gathered. This is an updatable column. For a sparse index, the statistic is based on the actual contents of the index.	S
CREATEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the CREATE statement was executed for the index. If the index was created in a DB2 release prior to Version 5, the value is '0001-01-01.00.00.00.000000'.	G
ALTEREDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the most recent ALTER INDEX statement was executed for the index. If no ALTER INDEX statement has been applied, ALTEREDTS has the value of CREATEDTS. If the index was created in a DB2 release prior to Version 5, the value is '0001-01-01.00.00.00.000000'.	G
PIECESIZE	INTEGER NOT NULL WITH DEFAULT	Maximum size of a data set in kilobytes for secondary indexes. A value of zero (0) indicates that the index is a partitioning index or that the index was created in a DB2 release prior to Version 5.	G
COPY	CHAR(1) NOT NULL WITH DEFAULT 'N'	Whether COPY YES was specified for the index, which indicates if the index can be copied and if SYSIBM.SYSLGRNX recording is enabled for the index. **N** No **Y** Yes	G
COPYLRSN	CHAR(10) NOT NULL WITH DEFAULT X'000000000000000000' FOR BIT DATA	The value can be either an RBA or LRSN. (LRSN is only for data sharing.) If the index is currently defined as COPY YES, the value is the RBA or LRSN when the index was created with COPY YES or altered to COPY YES, not the current RBA or LRSN. If the index is currently defined as COPY NO, the value is set to X'000000000000' if the index was created with COPY NO; otherwise, if the index was altered to COPY NO, the value in COPYLRSN is not changed when the index is altered to COPY NO.	G
CLUSTERRATIOF	FLOAT NOT NULL WITH DEFAULT	When multiplied by 100, the value of the column is the percentage of rows that are in clustering order. For example, a value of '.9125' indicates 91.25%. For a partitioning index, it is the weighted average of all index partitions in terms of the number of rows in the partition. The value is 0 if statistics have not been gathered. The value is -2 if the index is for an auxiliary table, a node ID index or an XML index. This is an updatable column. For a sparse index, the statistic is based on the actual contents of the index.	G
SPACEF	FLOAT(8) NOT NULL WITH DEFAULT -1	Kilobytes of DASD storage. The value is -1 if statistics have not been gathered. This is an updatable column.	G
REMARKS	VARCHAR(762) NOT NULL WITH DEFAULT	A character field string provided by the user with the COMMENT statement.	G

Column name	Data type	Description	Use
PADDED	CHAR(1) NOT NULL WITH DEFAULT	Indicates whether keys within the index are padded for varying-length column data: **Y** The index contains varying-length character or graphic data and is PADDED (the varying-length columns are padded to their maximum length). **N** The index contains varying-length character or graphic data and is NOT PADDED (the varying-length columns are not padded to their maximum length). Index-only access to all column data is possible. **blank** The index does not contain varying-length character or graphic data. The value is blank for indexes that have been created or altered prior to Version 8.	G
VERSION	SMALLINT NOT NULL WITH DEFAULT	The version of the data row format for this index. A value of zero indicates that a version-creating alter has never occurred against this index.	G
OLDEST_VERSION	SMALLINT NOT NULL WITH DEFAULT	The version number describing the oldest format of data in the index space and any image copies of the index.	G
CURRENT_VERSION	SMALLINT NOT NULL WITH DEFAULT	The version number describing the newest format of data in the index space. A zero indicates that the index space has never had been versioned. After the version number reaches the maximum value, the number will wrap back to one.	G
RELCREATED	CHAR(1) NOT NULL WITH DEFAULT	Release of DB2 that was used to create the object, blank for indexes created before Version 8. For all other values, see Release dependency indicators.	G
AVGKEYLEN	INTEGER NOT NULL WITH DEFAULT -1	Average length of keys within the index. The value is -1 if statistics have not been gathered. For a sparse index, the statistic is based on the actual contents of the index.	G
	CHAR(1) NOT NULL	Not used	N
KEYTARGET_COUNT	SMALLINT NOT NULL WITH DEFAULT	The number of key-targets for an extended index. The value is 0 for a simple index.	G
UNIQUE_COUNT	SMALLINT NOT NULL WITH DEFAULT	The number of columns or key-targets that make up the unique constraint of an index, when other non-constraint enforcing columns or key-targets exist. Otherwise the value is 0.	G
IX_EXTENSION_TYPE	CHAR(1) NOT NULL WITH DEFAULT	Identifies the type of extended index: **N** Node ID index **S** Index on a scalar expression **T** Spatial index **V** XML index **blank** Simple index	G

Column name	Data type	Description	Use
COMPRESS	CHAR(1) NOT NULL WITH DEFAULT ' N'	Indicates whether index compression is active: **N** Index compression is not active **Y** Index compression is active	G
OWNER	VARCHAR(128) NOT NULL WITH DEFAULT	Authorization ID of the owner of the index, empty string for indexes created in a DB2 release prior to Version 9.	G
OWNERTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of owner: **blank** Authorization ID **L** Role	G
DATAREPEAT-FACTORF	FLOAT NOT NULL WITH DEFAULT -1	The anticipated number of data pages that will be touched following an index key order. This number is -1 if statistics have not been collected. This is an updatable column. For a sparse index, the statistic is based on the actual contents of the index.	G
ENVID	INTEGER NOT NULL WITH DEFAULT	Internal environment identifier.	G
HASH	CHAR(1) NOT NULL WITH DEFAULT N	Whether the index is the hash overflow index for a hash table. **N** No. N is the default. **Y** Yes	G
SPARSE	CHAR(1) NOT NULL WITH DEFAULT N	Whether the index is sparse or not. **N** No. N is the default. Every data row has an index entry. **Y** Yes. This index might not have an entry for each data row in the table. **X** Excluded. This index will not have an index entry when every data row for a key column contains the NULL value.	G
ROWID	ROWID NOT NULL GENERATED ALWAYS	ROWID column, created for the lob columns in this table.	G
	BLOB(1G) NOT NULL WITH DEFAULT	Internal use only	I
	BLOB(1G) NOT NULL WITH DEFAULT	Internal use only	I

SYSIBM.SYSINDEXES_HIST table

The SYSIBM.SYSINDEXES_HIST table contains rows from SYSINDEXES.

Rows are added or changed in this table when RUNSTATS collects history statistics. Rows in this table can also be inserted, updated, and deleted.

Column name	Data type	Description	Use
NAME	VARCHAR(128) NOT NULL	Name of the index.	G
CREATOR	VARCHAR(128) NOT NULL	The schema of the index.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table on which the index is defined.	G
TBCREATOR	VARCHAR(128) NOT NULL	The schema of the table.	G
CLUSTERING	CHAR(1) NOT NULL	Whether CLUSTER was specified when the index was created: **N** No **Y** Yes	G
NLEAF	INTEGER NOT NULL WITH DEFAULT -1	Number of active leaf pages in the index. The value is -1 if statistics have not been gathered.	S
NLEVELS	SMALLINT NOT NULL WITH DEFAULT -1	Number of levels in the index tree. If the index is partitioned, it is the maximum of the number of levels in the index tree for all the partitions. The value is -1 if statistics have not been gathered.	S
STATSTIME	TIMESTAMP NOT NULL	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.00.000000'.	G
FIRSTKEYCARDF	FLOAT(8) NOT NULL WITH DEFAULT -1	Number of distinct values of the first key column. This number is an estimate if updated while collecting statistics on a single partition. The value is -1 if statistics have not been gathered.	S
FULLKEYCARDF	FLOAT(8) NOT NULL WITH DEFAULT -1	Number of distinct values of the key. The value is -1 if statistics have not been gathered.	S
CLUSTERRATIOF	FLOAT(8) NOT NULL	Percentage of rows that are in clustering order. For a partitioning index, it is the weighted average of all index partitions in terms of the number of rows in the partition. The value is 0 if statistics have not been gathered. The value is -2 if the index is for an auxiliary table. For a sparse index, the statistic is based on the actual contents of the index.	G
SPACEF	FLOAT(8) NOT NULL WITH DEFAULT -1	Number of kilobytes of DASD storage allocated to the index space partition. The value is -1 if statistics have not been gathered.	G

Column name	Data type	Description	Use
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
AVGKEYLEN	INTEGER NOT NULL WITH DEFAULT -1	Average length of keys within the index. The value is -1 if statistics have not been gathered. For a sparse index, the statistic is based on the actual contents of the index.	G
DATAREPEAT-FACTORF	FLOAT NOT NULL WITH DEFAULT -1	The anticipated number of data pages that will be touched following an index key order. This number is -1 if statistics have not been collected. This is an updatable column. For a sparse index, the statistic is based on the actual contents of the index.	G

SYSIBM.SYSINDEXES_RTSECT table

The SYSIBM.SYSINDEXES_RTSECT table is an auxiliary table for the RTSECTION column of the SYSIBM.SYSINDEXES table and is required to hold LOB data.

Column name	Data type	Description	Use
	BLOB(1G) NOT NULL WITH DEFAULT	Internal use only.	I

SYSIBM.SYSINDEXES_TREE table

The SYSIBM.SYSINDEXES_TREE table is an auxiliary table for the PARSETREE column of the SYSIBM.SYSINDEXES table and is required to hold LOB data.

Column name	Data type	Description	Use
	BLOB(1G) NOT NULL WITH DEFAULT	Internal use only.	I

SYSIBM.SYSINDEXPART table

The SYSIBM.SYSINDEXPART table contains one row for each nonpartitioned secondary index (NPSI) and one row for each partition of a partitioning index or a data-partitioned secondary index.

Column name	Data type	Description	Use
PARTITION	SMALLINT NOT NULL	Partition number; Zero if index is not partitioned.	G
IXNAME	VARCHAR(128) NOT NULL	Name of the index.	G
IXCREATOR	VARCHAR(128) NOT NULL	The schema of the index.	G
PQTY	INTEGER NOT NULL	For user-managed data sets, the value is the primary space allocation in units of 4KB storage blocks or -1. PQTY is based on a value of PRIQTY in the appropriate CREATE or ALTER INDEX statement. Unlike PQTY, however, PRIQTY asks for space in 1KB units. A value of -1 indicates that either of the following cases is true: • PRIQTY was not specified for a CREATE INDEX statement or for any subsequent ALTER INDEX statements. • -1 was the most recently specified value for PRIQTY, either on the CREATE INDEX statement or a subsequent ALTER INDEX statement.	G
SQTY	SMALLINT NOT NULL	For user-managed data sets, the value is the secondary space allocation in units of 4KB storage blocks or -1. SQTY is based on a value of SECQTY in the appropriate CREATE or ALTER INDEX statement. Unlike SQTY, however, SECQTY asks for space in 1KB units. A value of -1 indicates that either of the following cases is true: • SECQTY was not specified for a CREATE INDEX statement or for any subsequent ALTER INDEX statements. • -1 was the most recently specified value for SECQTY, either on the CREATE INDEX statement or a subsequent ALTER INDEX statement. If the value does not fit into the column, the value of the column is 32767. See the description of column SECQTYI.	G
STORTYPE	CHAR(1) NOT NULL	Type of storage allocation: **E** Explicit, and STORNAME names an integrated catalog facility catalog **I** Implicit, and STORNAME names a storage group	G
STORNAME	VARCHAR(128) NOT NULL	Name of storage group or integrated catalog facility catalog used for space allocation.Blank for the catalog indexes.	G

Column name	Data type	Description	Use
VCATNAME	VARCHAR(24) NOT NULL	Name of integrated catalog facility catalog used for space allocation.	G
CARD	INTEGER NOT NULL	Not used	N
FAROFFPOS	INTEGER NOT NULL	Not used	N
LEAFDIST	INTEGER NOT NULL	100 times the average number of leaf pages between successive active leaf pages of the index. The value is -1 if statistics have not been gathered. The value is -2 if the index is an auxiliary index, a node ID index, or an XML index.	S
NEAROFFPOS	INTEGER NOT NULL	Not used	S
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
LIMITKEY	VARCHAR(512) NOT NULL FOR BIT DATA	The high value of the limit key of the partition in an internal format. An empty string if the index is not partitioned or for a data-partitioned secondary index (DPSI). If any column of the key has a field procedure, the internal format is the encoded form of the value.	S
FREEPAGE	SMALLINT NOT NULL	Number of pages that are loaded before a page is left as free space.	G
PCTFREE	SMALLINT NOT NULL	Percentage of each leaf or nonleaf page that is left as free space.	G

Column name	Data type	Description	Use
SPACE	INTEGER NOT NULL WITH DEFAULT	Number of kilobytes of DASD storage allocated to the index space partition, as determined by the last execution of the STOSPACE utility. **0** The STOSPACE or RUNSTATS utility has not been run or the data set for the index has been created during the first insert operation or when the LOAD utility was run. **-1** The index was defined with the DEFINE NO clause, which defers the physical creation of the data sets until data is first inserted into the index, and data has yet to be inserted into the index. **A non-negative value** Indicates that the data sets for the index space are defined with the underlying data sets allocated. The value is updated by STOSPACE if the index is related to a storage group. The value is updated by RUNSTATS if the utility is executed as RUNSTATS INDEX with UPDATE(ALL) or UPDATE(SPACE).	G
STATSTIME	TIMESTAMP NOT NULL WITH DEFAULT	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.00.000000'.	G
	CHAR(1) NOT NULL	Not used	N
GBPCACHE	CHAR(1) NOT NULL WITH DEFAULT	Group buffer pool cache option specified for this index or index partition. **blank** Only changed pages are cached in the group buffer pool. **A** Changed and unchanged pages are cached in the group buffer pool. **N** No data is cached in the group buffer pool.	G
FAROFFPOSF	FLOAT NOT NULL WITH DEFAULT -1	Number of referred to rows far from optimal position because of an insert into a full page. The value is -1 if statistics have not been gathered. The value is -2 if the index is an auxiliary index, a node ID index, or an XML index. The column is not applicable for an index on an auxiliary table. For a sparse index, the statistic is based on the actual contents of the index.	S
NEAROFFPOSF	FLOAT NOT NULL WITH DEFAULT -1	Number of referred to rows near, but not at optimal position, because of an insert into a full page. The value is -2 if the index is an auxiliary index, a node ID index, or an XML index. Not applicable for an index on an auxiliary table. For a sparse index, the statistic is based on the actual contents of the index.	S

Column name	Data type	Description	Use
CARDF	FLOAT NOT NULL WITH DEFAULT -1	Number of RIDs in the index that refer to data rows or LOBs. The value is -1 if statistics have not been gathered. For a sparse index, the statistic is based on the actual contents of the index.	S
SECQTYI	INTEGER NOT NULL WITH DEFAULT	Secondary space allocation in units of 4KB storage. For user-managed data sets, the value is the secondary space allocation in units of 4KB blocks.	G
IPREFIX	CHAR(1) NOT NULL WITH DEFAULT 'I'	The first character of the instance qualifier for this index's data set name. 'I' or 'J' are the only valid characters for this field. The default is 'I'.	G
ALTEREDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the most recent ALTER INDEX statement was executed for the index. If no ALTER INDEX statement has been applied, the value is '0001-01-01.00.00.00.000000'.	G
SPACEF	FLOAT(8) NOT NULL WITH DEFAULT -1	Kilobytes of DASD storage. The value is -1 if statistics have not been gathered. This is an updatable column.	G
DSNUM	INTEGER NOT NULL WITH DEFAULT -1	Number of data sets. The value is -1 if statistics have not been gathered. This is an updatable column.	G
EXTENTS	INTEGER NOT NULL WITH DEFAULT -1	Number of data set extents. The value is -1 if statistics have not been gathered. This is an updatable column. This value is only for the last DSNUM for the object.	G
PSEUDO_DEL_ ENTRIES	INTEGER NOT NULL WITH DEFAULT -1	Number of pseudo deleted entries (entries that are logically deleted but still physically present in the index). For a non-unique index, value is the number of RIDs that are pseudo deleted. For a unique index, the value is the number of keys and RIDs that are pseudo deleted. The value is -1 if statistics have not been gathered. This is an updatable column.	G
LEAFNEAR	INTEGER NOT NULL WITH DEFAULT -1	Number of leaf pages physically near previous leaf page for successive active leaf pages. The value is -1 if statistics have not been gathered. This is an updatable column.	S
LEAFFAR	INTEGER NOT NULL WITH DEFAULT -1	Number of leaf pages located physically far away from previous leaf pages for successive (active leaf) pages accessed in an index scan. The value is -1 if statistics have not been gathered. This is an updatable column.	S
OLDEST_VERSION	SMALLINT NOT NULL WITH DEFAULT	The version number describing the oldest format of data in the index part and any image copies of the index part.	G
CREATEDTS	TIMESTAMP NOT NULL WITH DEFAULT -1	Time when the partition was created.	G

Column name	Data type	Description		Use
AVGKEYLEN	INTEGER NOT NULL WITH DEFAULT -1	Average length of keys within the index. The value is -1 if statistics have not been gathered. For a sparse index, the statistic is based on the actual contents of the index.		G
RBA_FORMAT	CHAR(1) NOT NULL WITH DEFAULT	Indicates the format of the RBA/LRSN.		G
		B	Basic, 6-byte RBA/LRSN format.	
		E	Extended, 10-byte RBA/LRSN format.	
		U	Undefined. DEFINE NO was specified when creating the table space.	
		blank	For migrated objects.	

SYSIBM.SYSINDEXPART_HIST table

The SYSIBM.SYSINDEXPART_HIST table contains rows from SYSINDEXPART.

Rows are added or changed in this table when RUNSTATS collects history statistics. Rows in this table can also be inserted, updated, and deleted.

Column name	Data type	Description	Use
PARTITION	SMALLINT NOT NULL	Partition number. Zero if index is not partitioned.	G
IXNAME	VARCHAR(128) NOT NULL	Name of the index.	G
IXCREATOR	VARCHAR(128) NOT NULL	The schema of the index.	G
PQTY	INTEGER NOT NULL	For user-managed data sets, the value is the primary space allocation in units of 4KB storage blocks or -1. For user-specified values of PRIQTY other than -1, the value is set to the primary space allocation only if RUNSTATS INDEX with UPDATE(ALL) or UPDATE(SPACE) is executed; otherwise, the value is zero. PQTY is based on a value of PRIQTY in the appropriate CREATE or ALTER INDEX statement. Unlike PQTY, however, PRIQTY asks for space in 1KB units. A value of -1 indicates that either of the following cases is true: • PRIQTY was not specified for a CREATE INDEX statement or for any subsequent ALTER INDEX statements. • -1 was the most recently specified value for PRIQTY, either on the CREATE INDEX statement or a subsequent ALTER INDEX statement. If a storage group is not used, the value is 0.	G
SECQTYI	INTEGER NOT NULL	For user-managed data sets, the value is the secondary space allocation in units of 4KB storage blocks or -1. For user-specified values of SECQTY other than -1, the value is set to the secondary space allocation only if RUNSTATS INDEX with UPDATE(ALL) or UPDATE(SPACE) is executed; otherwise, the value is zero. SQTY is based on a value of SECQTY in the appropriate CREATE or ALTER INDEX statement. Unlike SQTY, however, SECQTY asks for space in 1KB units. A value of -1 indicates that either of the following cases is true: • SECQTY was not specified for a CREATE INDEX statement or for any subsequent ALTER INDEX statements. • -1 was the most recently specified value for SECQTY, either on the CREATE INDEX statement or a subsequent ALTER INDEX statement. If a storage group is not used, the value is 0.	G

Column name	Data type	Description	Use
LEAFDIST	INTEGER NOT NULL WITH DEFAULT -1	100 times the average number of leaf pages between successive active leaf pages of the index. The value is -1 if statistics have not been gathered.	S
SPACEF	FLOAT(8) NOT NULL WITH DEFAULT -1	Number of kilobytes of DASD storage allocated to the index space partition. The value is -1 if statistics have not been gathered.	G
STATSTIME	TIMESTAMP NOT NULL	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.00.000000'.	G
FAROFFPOSF	FLOAT(8) NOT NULL WITH DEFAULT -1	Number of referred to rows far from optimal position because of an insert into a full page. The value is -1 if statistics have not been gathered. The column is not applicable for an index on an auxiliary table. For a sparse index, the statistic is based on the actual contents of the index.	S
NEAROFFPOSF	FLOAT(8) NOT NULL WITH DEFAULT -1	Number of referred to rows near, but not at optimal position, because of an insert into a full page. Not applicable for an index on an auxiliary table. The value is -1 if statistics have not been gathered. For a sparse index, the statistic is based on the actual contents of the index.	S
CARDF	FLOAT(8) NOT NULL WITH DEFAULT -1	Number of RIDs in the index that refer to data rows or LOBs. The value is -1 if statistics have not been gathered. For a sparse index, the statistic is based on the actual contents of the index.	S
EXTENTS	INTEGER NOT NULL WITH DEFAULT -1	Number of data set extents. The value is -1 if statistics have not been gathered. This value is only for the last DSNUM for the object.	G
PSEUDO_DEL_ ENTRIES	INTEGER NOT NULL WITH DEFAULT -1	Number of pseudo deleted entries. The value is -1 if statistics have not been gathered.	G
DSNUM	INTEGER NOT NULL WITH DEFAULT -1	Data set number within the table space. For partitioned index spaces, this value corresponds to the partition number for a single partition copy, or 0 for a copy of an entire partitioned index space. The value is -1 if statistics have not been gathered.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
LEAFNEAR	INTEGER NOT NULL WITH DEFAULT -1	Number of leaf pages physically near previous leaf page for successive active leaf pages. The value is -1 if statistics have not been gathered. This is an updatable column.	S

Column name	Data type	Description	Use
LEAFFAR	INTEGER NOT NULL WITH DEFAULT -1	Number of leaf pages located physically far away from previous leaf pages for successive (active leaf) pages accessed in an index scan. The value is -1 if statistics have not been gathered. This is an updatable column.	S
AVGKEYLEN	INTEGER NOT NULL WITH DEFAULT -1	Average length of keys within the index. The value is -1 if statistics have not been gathered. For a sparse index, the statistic is based on the actual contents of the index.	G

SYSIBM.SYSINDEXSPACESTATS table

The SYSIBM.SYSINDEXSPACESTATS table contains real time statistics for index spaces.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
UPDATESTATSTIME	TIMESTAMP NOT NULL WITH DEFAULT	The timestamp that the row in the SYSINDEXSPACESTATS table is inserted or last updated.	G
NLEVELS	SMALLINT	The number of levels in the index tree. A null value indicates that the number of levels is unknown.	G
NPAGES	INTEGER	The number of pages in the index tree that contain only pseudo-deleted index entries. This is an updatable column.	G
NLEAF	INTEGER	The number of leaf pages in the index. This is an updatable column.	G
NACTIVE	INTEGER	The number of active pages in the index space or partition. This value is equivalent to the number of pre-formatted pages. A null value indicates that the number of active pages is unknown.	G
SPACE	INTEGER	The amount of space, in KB, that is allocated to the index space or partition. For multi-piece, linear page sets, this value is the amount of space in all data sets. A null value indicates the amount of space is unknown.	G
EXTENTS	SMALLINT	The number of extents in the index space or partition. For multi-piece index spaces, this value is the number of extents for the last data sets. For a data set that is stripped across multiple volumes, the value is the number of logical extents. A null value indicates the number of extents is unknown.	G
LOADRLASTTIME	TIMESTAMP	The timestamp that the LOAD REPLACE utility was last run on the index space or partition. A null value indicates that the LOAD REPLACE utility has never been run on the index space or partition or that the timestamp is unknown.	G
REBUILDLASTTIME	TIMESTAMP	The timestamp that the REBUILD INDEX utility was last run on the index space or partition. A null value indicates that the timestamp that the REBUILD INDEX was last run is unknown.	G
REORGLASTTIME	TIMESTAMP	The timestamp when the REORG INDEX utility was last run on the index space or partition, or if the REORG INDEX utility has not been run, the time when the index space or partition was created. A null value indicates that the timestamp is unknown.	G

Column name	Data type	Description	Use
REORGINSERTS	INTEGER	The number of index entries that have been inserted into the index space or partition since the last time the REORG, REBUILD INDEX, or LOAD REPLACE utilities were run, or since the object was created. A null value indicates that the number of inserted index entries is unknown.	G
REORGDELETES	INTEGER	The number of index entries that have been deleted from the index space or partition since the last time the REORG, REBUILD INDEX, or LOAD REPLACE utilities were run, or since the object was created. A null value indicates that the number of deleted index entries is unknown.	G
REORGAPPEND-INSERT	INTEGER	The number of index entries that have a key value that is greater than the maximum key value in the index or partition that have been inserted into the index space or partition since the last time the REORG, REBUILD INDEX, or LOAD REPLACE utilities were run, or since the object was created. A null value indicates that the number of inserted index entries is unknown.	G
REORGPSEUDO-DELETES	INTEGER	The number of pseudo-deleted index entries stored in the index space or partition. A pseudo-delete is a RID entry that has been marked as deleted. A null value indicates that the number of pseudo-deleted index entries is unknown.	G
REORGMASSDELETE	INTEGER	The number of mass deletes from a segmented or LOB table space, or the number of dropped tables from a segmented table space since the last time the REORG or LOAD REPLACE utilities were run, or since the object was created. A null value indicates that the number of mass deletes is unknown.	G

Column name	Data type	Description	Use
REORGLEAFNEAR	INTEGER	The net number of leaf pages located physically near previous pages for successive active leaf pages that occurred since the last REORG, REBUILD INDEX, or LOAD REPLACE, or since the object was created. The distance between leaf pages is optimal if the difference is 1 and considered near if the distance is 2-16. An index page is added during a page split and the distance between the predecessor and successor pages can lower this count if the distance between the two was near. The distance between the predecessor and new page increase the count if they are near. The distance between the new page and successor increment the count if they are near. If a leaf page is deleted the distance between the new predecessor and successor pages can increment this count if the distance between the two is near. The distance between the predecessor and the deleted page decrement the count if it was near. The distance between the successor and the deleted page decrement the count if it was near. A null value means that the value is unknown. A negative value is possible in some cases.	G
REORGLEAFFAR	INTEGER	The net number of leaf pages located physically far away from previous leaf pages for successive active leaf pages that occurred since the last REORG, REBUILD INDEX, or LOAD REPLACE, or since the object was created. The distance between leaf pages is optimal if the difference is 1 and considered far if the distance is greater than 16. An index page is added during a page split and the distance between the predecessor and successor pages can decrement this count if the distance between the two was far. The distance between the predecessor and new page increment the count if they are far. The distance between the new page and successor increment the count if they are far. If a leaf page is deleted the distance between the new predecessor and successor pages can increment this count if the distance between the two is far. The distance between the predecessor and the deleted page decrement the count if it was far. The distance between the successor and the deleted page decrement the count if it was far. A null value means that the value is unknown.	G
REORGNUMLEVELS	INTEGER	The number of levels in the index tree that were added or removed since the last REORG, REBUILD INDEX, or LOAD REPLACE, or the object was created. A null value means that the number of added or deleted levels is unknown.	G

Column name	Data type	Description	Use
STATSLASTTIME	TIMESTAMP	The timestamp of the last time that the RUNSTATS utility is run on the table space or partition. A null value means that RUNSTATS has never been run on the index space or partition, or that the timestamp of the last RUNSTATS is unknown.	G
STATSINSERTS	INTEGER	The number of index entries that have been inserted into the index space or partition since the last time that the RUNSTATS utility was run, or since the object was created. A null value indicates that the number of inserted records or LOBs is unknown.	G
STATSDELETES	INTEGER	The number of index entries that have been deleted since the last RUNSTATS on the index space or partition, or since the object was created. A null value means that the number of deleted index entries is unknown.	G
STATSMASSDELETE	INTEGER	The number of times that the index or index space partition was mass deleted since the last RUNSTATS, or the object was created. A null value indicates that the number of mass deletes is unknown.	G
COPYLASTTIME	TIMESTAMP	The timestamp of the last full image copy on the index space or partition. A null value means that COPY has never been run on the index space or partition, or that the timestamp of the last full image copy is unknown.	G
COPYUPDATED-PAGES	INTEGER	The number of distinct pages that have been updated since the last time that the COPY utility was run, or since the object was created. A null value indicates that the number of updated pages is unknown.	G
COPYCHANGES	INTEGER	The number of insert, update, and delete operations since the last time that the COPY utility was run, or since the object was created. A null value indicates that the number of insert, update, and delete operations is unknown.	G
COPYUPDATELRSN	CHAR(10) FOR BIT DATA	The LRSN or RBA of the first update that occurs after the last time the COPY utility was run. A null value indicates that the LRSN or RBA is unknown.	G
COPYUPDATETIME	TIMESTAMP	The timestamp of the first update that occurs after the last time that the COPY utility was run. A null value indicates that the timestamp is unknown.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

Column name	Data type	Description	Use
DBID	SMALLINT NOT NULL	The internal identifier of the database.	G
ISOBID	SMALLINT NOT NULL	The internal identifier of the index space page set descriptor.	I
PSID	SMALLINT NOT NULL	The internal identifier of the table space page set descriptor for the table space that is associated with the index.	G
PARTITION	SMALLINT NOT NULL	The data set number within the index space. For partitioned index spaces, this value corresponds to the partition number for a single partition. For non-partitioned table spaces, this value is 0.	G
INSTANCE	SMALLINT NOT NULL WITH DEFAULT 1	Indicates if the object is associated with data set 1 or 2. This is an updatable column.	G
TOTALENTRIES	BIGINT	The number of entries, including duplicate entries, in the index space or partition. A null value indicates that the number of entries is unknown.	G
DBNAME	VARCHAR(24) NOT NULL	The name of the database.	G
NAME	VARCHAR(128) NOT NULL	The name of the index.	G
CREATOR	VARCHAR(128) NOT NULL	The schema of the index.	G
INDEXSPACE	VARCHAR(24) NOT NULL	The name of the index space.	G
LASTUSED	DATE	The date when the index is used for SELECT, FETCH, searched UPDATE, searched DELETE, or used to enforce referential integrity constraints. For a data-partitioned secondary index, this column is only updated for one partition, even though more than one partition is accessed. The default value is NULL.	G
REORGINDEXACCESS	BIGINT	The number of times the index was used for SELECT, FETCH, searched UPDATE, searched DELETE, or used to enforce referential integrity constraints, or since the object was created. For hash overflow indexes, this is the number of times DB2 has used the hash overflow index. A null value indicates that the number of times the index has been used is unknown.	G

Column name	Data type	Description	Use
DRIVETYPE	CHAR(3) NOT NULL WITH DEFAULT	The drive type on which the index or index partition data set is defined. **HDD** Hard Disk Drive **SSD** Solid State Drive For multi-volume data sets, the drive type is set to SSD if any volume is SSD. For multi-piece linear page sets, the drive type of the first data set is used.	G
	BIGINT	Reserved for future IBM use.	R

In data sharing environments, the values in SYSIBM.SYSINDEXSPACESTATS can be negative for short periods of time for certain situations.

Related concepts:

How DB2 maintains in-memory statistics in data sharing (DB2 Data Sharing Planning and Administration)

SYSIBM.SYSINDEXSTATS table

The SYSIBM.SYSINDEXSTATS table contains one row for each partition of a partitioning index or a data-partitioned secondary index.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
FIRSTKEYCARD	INTEGER NOT NULL	For the index partition, number of distinct values of the first key column. For a sparse index, the statistic is based on the actual contents of the index.	S
FULLKEYCARD	INTEGER NOT NULL	For the index partition, number of distinct values of the key. For a sparse index, the statistic is based on the actual contents of the index.	S
NLEAF	INTEGER NOT NULL	Number of active leaf pages in the index partition.	S
NLEVELS	SMALLINT NOT NULL	Number of levels in the index tree.	S
	SMALLINT NOT NULL	Not used	N
	SMALLINT NOT NULL	Not used	N
CLUSTERRATIO	SMALLINT NOT NULL	For the index partition, the percentage of rows that are in clustering order. The value is 0 if statistics have not been gathered. For a sparse index, the statistic is based on the actual contents of the index.	N
STATSTIME	TIMESTAMP NOT NULL	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.00.000000'.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
PARTITION	SMALLINT NOT NULL	Partition number of the index.	G
OWNER	VARCHAR(128) NOT NULL	The schema of the index.	G
NAME	VARCHAR(128) NOT NULL	Name of the index.	G

Column name	Data type	Description	Use
KEYCOUNT	INTEGER NOT NULL	Total number of RIDs in the index partition. The value is -1 if statistics have not been gathered.Total number of rows in the partition. For a sparse index, the statistic is based on the actual contents of the index.	S
FIRSTKEYCARDF	FLOAT NOT NULL WITH DEFAULT -1	For the index partition, number of distinct values of the first key column. For a sparse index, the statistic is based on the actual contents of the index.	S
FULLKEYCARDF	FLOAT NOT NULL WITH DEFAULT -1	For the index partition, number of distinct values of the key. For a sparse index, the statistic is based on the actual contents of the index.	S
KEYCOUNTF	FLOAT WITH DEFAULT -1	Total number of RIDs in the index partition. The value is -1 if statistics have not been gathered.Total number of rows in the partition. For a sparse index, the statistic is based on the actual contents of the index.	S
CLUSTERRATIOF	FLOAT NOT NULL WITH DEFAULT	For the index partition, the value, when multiplied by 100, is the percentage of rows that are in clustering order. For example, a value of '.9125' indicates 91.25%. The value is 0 if statistics have not been gathered. For a sparse index, the statistic is based on the actual contents of the index.	G
	VARCHAR(1000) NOT NULL WITH DEFAULT FOR BIT DATA	Internal use only	I
DATAREPEAT-FACTORF	FLOAT NOT NULL WITH DEFAULT -1	The anticipated number of data pages that will be touched following an index key order. This number is -1 if statistics have not been collected. This is an updatable column. For a sparse index, the statistic is based on the actual contents of the index.	G

SYSIBM.SYSINDEXSTATS_HIST table

The SYSIBM.SYSINDEXSTATS_HIST table contains rows from SYSINDEXSTATS.

Rows are added or changed in this table when RUNSTATS collects history statistics. Rows in this table can also be inserted, updated, and deleted.

Column name	Data type	Description	Use
NLEAF	INTEGER NOT NULL WITH DEFAULT -1	Number of active leaf pages in the index partition. The value is -1 if statistics have not been gathered.	S
NLEVELS	SMALLINT NOT NULL WITH DEFAULT -1	Number of levels in the index tree. The value is -1 if statistics have not been gathered.	S
STATSTIME	TIMESTAMP NOT NULL	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.00.000000'.	G
PARTITION	SMALLINT NOT NULL	Partition number of the index.	G
OWNER	VARCHAR(128) NOT NULL	The schema of the index.	G
NAME	VARCHAR(128) NOT NULL	Name of the index.	G
FIRSTKEYCARDF	FLOAT(8) NOT NULL WITH DEFAULT -1	For the index partition, number of distinct values of the first key column. The value is -1 if statistics have not been gathered.	S
FULLKEYCARDF	FLOAT(8) NOT NULL WITH DEFAULT -1	For the index partition, number of distinct values of the key. The value is -1 if statistics have not been gathered.	S
KEYCOUNTF	FLOAT(8) NOT NULL WITH DEFAULT -1	Total number of RIDs in the index partition. The value is -1 if statistics have not been gathered.Total number of rows in the partition. The value is -1 if statistics have not been gathered. For a sparse index, the statistic is based on the actual contents of the index.	S
CLUSTERRATIOF	FLOAT(8) NOT NULL	For the index partition, the value, when multiplied by 100, is the percentage of rows that are in clustering order. For example, a value of '0.9125' indicates 91.25%. The value is 0 if statistics have not been gathered. For a sparse index, the statistic is based on the actual contents of the index.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

Column name	Data type	Description	Use
DATAREPEAT-FACTORF	FLOAT NOT NULL WITH DEFAULT -1	The anticipated number of data pages that will be touched following an index key order. This number is -1 if statistics have not been collected. This is an updatable column. For a sparse index, the statistic is based on the actual contents of the index.	G

SYSIBM.SYSJARCLASS_SOURCE table

The SYSIBM.SYSJARCLASS_SOURCE table is an auxiliary table for SYSIBM.SYSJARCONTENTS.

Column name	Data type	Description	Use
CLASS_SOURCE	CLOB(10M) NOT NULL	The contents of the class in the JAR file.	G

SYSIBM.SYSJARCONTENTS table

The SYSIBM.SYSJARCONTENTS table contains Java class source for an installed JAR file.

Column name	Data type	Description	Use
JARSCHEMA	VARCHAR(128) NOT NULL	The schema of the JAR file.	G
JAR_ID	VARCHAR(128) NOT NULL	The name of the JAR file.	G
CLASS	VARCHAR(384) NOT NULL	The class name contained in the JAR file.	G
CLASS_SOURCE_ROWID	ROWID NOT NULL GENERATED ALWAYS	ID used to support CLOB data type.	G
CLASS_SOURCE	CLOB(10M) NOT NULL	The contents of the class in the JAR file.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

SYSIBM.SYSJARDATA table

The SYSIBM.SYSJARDATA table is an auxiliary table for SYSIBM.SYSJAROBJECTS.

Column name	Data type	Description	Use
JAR_DATA	BLOB(100M) NOT NULL	The contents of the JAR file.	G

SYSIBM.SYSJAROBJECTS table

The SYSIBM.SYSJAROBJECTS table contains binary large object representing the installed JAR file.

Column name	Data type	Description	Use
JARSCHEMA	VARCHAR(128) NOT NULL	The schema of the JAR file.	G
JAR_ID	VARCHAR(128) NOT NULL	The name of the JAR file.	G
OWNER	VARCHAR(128) NOT NULL	Authorization ID of the owner of the JAR object.	G
JAR_DATA_ROWID	ROWID NOT NULL GENERATED ALWAYS	ID used to support BLOB data type.	G
JAR_DATA	BLOB(100M) NOT NULL	The contents of the JAR file. This is an updatable column.	G
PATH	VARCHAR(2048) NOT NULL	The class resolution path of the JAR file. This is an updatable column.	G
CREATEDTS	TIMESTAMP NOT NULL	Time when the JAR object was created.	G
ALTEREDTS	TIMESTAMP NOT NULL	Time when the JAR object was altered.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
OWNERTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of owner: **blank** Authorization ID **L** Role	G

SYSIBM.SYSJAVAOPTS table

The SYSIBM.SYSJAVAOPTS table contains build options used during INSTALL_JAR.

Column name	Data type	Description	Use
JARSCHEMA	VARCHAR(128) NOT NULL	The schema of the JAR file.	G
JAR_ID	VARCHAR(128) NOT NULL	The name of the JAR file.	G
BUILDSCHEMA	VARCHAR(128) NOT NULL	Schema name for BUILDNAME.	G
BUILDNAME	VARCHAR(128) NOT NULL	Procedure used to create the routine.	G
BUILDOWNER	VARCHAR(128) NOT NULL	Authorization ID used to create the routine.	G
DBRMLIB	VARCHAR(256) NOT NULL	PDS name where DBRM is located.	G
HPJCOMPILE_OPTS	VARCHAR(512) NOT NULL	HPJ compile options used to install the routine.	G
BIND_OPTS	VARCHAR(2048) NOT NULL	Bind options used to install the routine.	G
POBJECT_LIB	VARCHAR(256) NOT NULL	PDSE name where program object is located.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

SYSIBM.SYSJAVAPATHS table

The SYSIBM.SYSJAVAPATHS table contains the complete class resolution path of a JAR file, and records the dependencies that one JAR file has on the JAR files in its Java path.

Column name	Data type	Description	Use
JARSCHEMA	VARCHAR(128) NOT NULL	The schema of the JAR file.	G
JAR_ID	VARCHAR(128) NOT NULL	The name of the JAR file.	G
OWNER	VARCHAR(128) NOT NULL	Authorization ID of the owner of the JAR object.	G
ORDINAL	SMALLINT NOT NULL	The ordinal number of the path element within the JAR file's Java path.	G
PE_CLASS_PATTERN	VARCHAR(2048) NOT NULL	The pattern for the names of the classes that are to be searched for in this path element's JAR file.	G
PE_JARSCHEMA	VARCHAR(128) NOT NULL	The schema of this path element's JAR file.	G
PE_JAR_ID	VARCHAR(128) NOT NULL	The name of this path element's JAR file.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

SYSIBM.SYSKEYCOLUSE table

The SYSIBM.SYSKEYCOLUSE table contains a row for every column in a unique constraint (primary key or unique key) from the SYSIBM.SYSTABCONST table.

Column name	Data type	Description	Use
CONSTNAME	VARCHAR(128) NOT NULL	Name of the constraint.	G
TBCREATOR	VARCHAR(128) NOT NULL	Schema or qualifier of the table on which the constraint is defined.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table on which the constraint is defined.	G
COLNAME	VARCHAR(128) NOT NULL	Name of the column	G
COLSEQ	SMALLINT NOT NULL	Numeric position of the column in the key (the first position in the key is 1).	G
COLNO	SMALLINT NOT NULL	Numeric position of the column in the table on which the constraint is defined.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
PERIOD	CHAR(1) NOT NULL WITH DEFAULT	Indicates whether the column is the start or end column for the BUSINESS_TIME period: **B** The column is the start of the period BUSINESS_TIME. **C** The column is the end of the period BUSINESS_TIME. **blank** Column is not used as either the start or the end of a BUSINESS_TIME period.	G

SYSIBM.SYSKEYS table

The SYSIBM.SYSKEYS table contains one row for each column of an index key.

Column name	Data type	Description	Use
IXNAME	VARCHAR(128) NOT NULL	Name of the index.	G
IXCREATOR	VARCHAR(128) NOT NULL	Schema or qualifier of the index.	G
COLNAME	VARCHAR(128) NOT NULL	Name of the column of the key.	G
COLNO	SMALLINT NOT NULL	Numeric position of the column in the table. For example, 4 (out of 10).	G
COLSEQ	SMALLINT NOT NULL	Numeric position of the column in the key for an index on columns. For example, 4 (out of 4). The value is meaningless for an expression-based indexes.	G
ORDERING	CHAR(1) NOT NULL	Order of the column in the key: **blank** Index is an expression-based index or the column is specified for the index using the INCLUDE clause **A** Ascending order **D** Descending order **R** Random order	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
PERIOD	CHAR(1) NOT NULL WITH DEFAULT	Indicates whether the column is the start or end column for the BUSINESS_TIME period: **B** The column is the start of the period BUSINESS_TIME. **C** The column is the end of the period BUSINESS_TIME. **blank** Column is not used as either the start or the end of a BUSINESS_TIME period.	G

Related concepts:

Index keys (Introduction to DB2 for z/OS)

SYSIBM.SYSKEYTARGETS table

The SYSIBM.SYSKEYTARGETS table contains one row for each key-target that is participating in an extended index definition.

Column name	Data type	Description	Use
IXNAME	VARCHAR(128) NOT NULL	Name of the index.	G
IXSCHEMA	VARCHAR(128) NOT NULL	Qualifier of the index.	G
KEYSEQ	SMALLINT NOT NULL	Numeric position of the key-target in the index.	G
COLNO	SMALLINT NOT NULL	Numeric position of the column in the table if the expression is a single column. Otherwise the value is 0. For XML indexes, this field is also 0.	G
ORDERING	CHAR(1) NOT NULL	Order of the key: **A** Ascending	G
TYPESCHEMA	VARCHAR(128) NOT NULL	Schema of the data type.	G
TYPENAME	VARCHAR(128) NOT NULL	Name of the data type.	G
DATATYPEID	INTEGER NOT NULL	The internal ID of the data type.	G
SOURCETYPEID	INTEGER NOT NULL	For a built-in data type, this column contains 0. For a distinct type, this column contains the internal ID of the built-in type on which the distinct type is based.	G

Column name	Data type	Description	Use
LENGTH	SMALLINT NOT NULL	The length attribute of the key-target or its precision for a decimal key-target. The number does not include the internal prefixes that are used to record the actual length and null states, when applicable. **data type** **value of the LENGTH column** **INTEGER** 4 **SMALLINT** 2 **FLOAT** 4 or 8 **CHAR** The length of the string **VARCHAR** The maximum length of the string **DECIMAL** The precision of the number **GRAPHIC** The number of DBCS characters **VARGRAPHIC** The maximum number of DBCS characters **DATE** 4 **TIME** 3 **TIMESTAMP WITHOUT TIME ZONE** The integral part of ((p+1)/2) + 7 where p is the precision of the timestamp **TIMESTAMP WITH TIME ZONE** The integral part of ((p+1)/2) + 9 where p is the precision of the timestamp **BIGINT** 8 **BINARY** The length of the string **VARBINARY** The maximum length of the string **DECFLOAT** 8 or 16	G
LENGTH2	INTEGER NOT NULL	The maximum length of the data that is retrieved from the column. Possible values include the following values: **0** Not a ROWID column **40** For a ROWID column, the length of the value that is returned	G
SCALE	SMALLINT NOT NULL	The scale of decimal data or number of fractional second digits of timestamp or timestamp with time zone data. Otherwise the value is 0. If the column is a timestamp type, the LENGTH is 10 and the SCALE is 0, the number of fractional second digits is 6.	G
NULLS	CHAR(1) NOT NULL	Whether the key can contain null values: **N** No **Y** Yes. Y also indicates that the index is an XML index.	G
CCSID	INTEGER NOT NULL	The CCSID of the key. CCSID contains 0 if the key is a non-character type key.	G

Column name	Data type	Description	Use
SUBTYPE	CHAR(1) NOT NULL	SUBTYPE applies to character keys only and indicated the subtype of the data: **B** BIT data **M** MIXED data **S** SBCS data **blank** non-character data	G
	VARCHAR(512) NOT NULL FOR BIT DATA	Internal use.	I
CREATEDTS	TIMESTAMP NOT NULL	The timestamp for when the key-target is created.	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 in which the key-target is created. SeeRelease dependency indicators for values.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies. RELCREATED should be used instead.	G
DERIVED_FROM	VARCHAR(4000) NOT NULL	For an index on a scalar expression, DERIVED_FROM contains the text of the scalar expression that is used to generated the key-target value. For an XML index, this is the XML pattern that is used to generate the key-target value. Otherwise DERIVED_FROM contains an empty string.	G
STATSTIME	TIMESTAMP NOT NULL WITH DEFAULT	The timestamp of the most recent RUNSTATS. The default value is '0001-01-01.00.00.00.000000'. STATSTIME is an updatable column.	G
CARDF	FLOAT NOT NULL WITH DEFAULT -1	The estimated number of distinct values for the key-target. The value is -2 if the index is a node ID index. For an XML value index, the statistic is collected for the second key target (the DOCID column). For all other key targets of the XML value index, a value of -2 is set.	G
HIGH2KEY	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	The second highest key-value. HIGH2KEY is an updatable column.	G
LOW2KEY	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	The second lowest key-value. LOW2KEY is an updatable column.	G

Column name	Data type	Description	Use
STATS_FORMAT	CHAR(1) NOT NULL WITH DEFAULT	The type of statistics that are gathered: **N** VARCHAR column statistical values are not padded **blank** Statistics have not been collects or VARCHAR column statistical values are padded STATS_FORMAT is an updatable column.	G

SYSIBM.SYSKEYTARGETSTATS table

The SYSIBM.SYSKEYTARGETSTATS table contains partition statistics for selected key-targets. For each key-target, a row exists for each partition in the table.

Rows are inserted when RUNSTATS collects indexed key statistics or non-indexed key statistics for a partitioned table space. No row is inserted if the table space is nonpartitioned. Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
IXSCHEMA	VARCHAR(128) NOT NULL	The qualifier of the index.	G
IXNAME	VARCHAR(128) NOT NULL	The name of the index.	G
KEYSEQ	SMALLINT NOT NULL	Numeric position of the key-target in the index.	G
HIGHKEY	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	The highest key value.	S
HIGH2KEY	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	The second highest key-value.	S
LOWKEY	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	The lowest key value.	S
LOW2KEY	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	The second lowest key-value.	S
PARTITION	SMALLINT NOT NULL	The partition number of the table space.	G
	VARCHAR(1000) NOT NULL WITH DEFAULT FOR BIT DATA	Internal use only.	I
STATSTIME	TIMESTAMP NOT NULL WITH DEFAULT	The timestamp of the most recent RUNSTATS. The default value is '0001-01-01.00.00.00.000000'.	G

Column name	Data type	Description	Use
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
STATS_FORMAT	CHAR(1) NOT NULL WITH DEFAULT	The type of statistics that are gathered: **N** VARCHAR column statistical values are not padded **blank** Statistics have not been collects or VARCHAR column statistical values are padded	G
CARDF	FLOAT NOT NULL WITH DEFAULT -1	Number of distinct values for the key target.	G

SYSIBM.SYSKEYTARGETS_HIST table

The SYSIBM.SYSKEYTARGETS_HIST table contains rows from the SYSKEYTARGETS table.

Whenever rows are added or changed in SYSKEYTARGETS, the rows are also written to this table. Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
IXNAME	VARCHAR(128) NOT NULL	Name of the index.	G
IXSCHEMA	VARCHAR(128) NOT NULL	Qualifier of the index.	G
KEYSEQ	SMALLINT NOT NULL	Numeric position of the key-target in the index.	G
TYPESCHEMA	VARCHAR(128) NOT NULL	Schema of the data type.	G
TYPENAME	VARCHAR(128) NOT NULL	Name of the data type.	G
DATATYPEID	INTEGER NOT NULL	The internal ID of the data type.	G
SOURCETYPEID	INTEGER NOT NULL	For a built-in data type, this field contains 0. For a distinct type, this field contains the internal ID of the built-in type on which the distinct type is based.	G

Column name	Data type	Description	Use
LENGTH	SMALLINT NOT NULL	The length attribute of the key-target or its precision for a decimal key-target. The number does not include the internal prefixes that are used to record the actual length and null states, when applicable. **data type** **value of the LENGTH column** **INTEGER** 4 **SMALLINT** 2 **FLOAT** 4 or 8 **CHAR** The length of the string **VARCHAR** The maximum length of the string **DECIMAL** The precision of the number **GRAPHIC** The number of DBCS characters **VARGRAPHIC** The maximum number of DBCS characters **DATE** 4 **TIME** 3 **TIMESTAMP WITHOUT TIME ZONE** The integral part of ((p+1)/2) + 7 where p is the precision of the timestamp **TIMESTAMP WITH TIME ZONE** The integral part of ((p+1)/2) + 9 where p is the precision of the timestamp **BIGINT** 8 **BINARY** The length of the string **VARBINARY** The maximum length of the string **DECFLOAT** 8 or 16	G
LENGTH2	INTEGER NOT NULL	The maximum length of the data that is retrieved from the column. Possible values include the following values: **0** Not a ROWID column **40** For a ROWID column, the length of the value that is returned	G
SCALE	SMALLINT NOT NULL	The scale of decimal data or number of fractional second digits of timestamp or timestamp with time zone data. Otherwise the value is 0. If the column is a timestamp type, the LENGTH is 10 and the SCALE is 0, the number of fractional second digits is 6.	G
NULLS	CHAR(1) NOT NULL	Whether the key can contain null values: **N** No **Y** Yes	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

Column name	Data type	Description	Use
STATSTIME	TIMESTAMP NOT NULL WITH DEFAULT	The timestamp of the most recent RUNSTATS. The default value is '0001-01-01.00.00.00.000000'. STATSTIME is an updatable column.	G
CARDF	FLOAT NOT NULL WITH DEFAULT -1	The estimated number of distinct values for the key-target. The value is -2 if the index is a node ID index. For an XML value index, the statistic is collected for the second key target (the DOCID column). For all other key targets of the XML value index, a value of -2 is set.	G
HIGH2KEY	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	The second highest key-value.	G
LOW2KEY	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	The second lowest key-value.	G
STATS_FORMAT	CHAR(1) NOT NULL WITH DEFAULT	The type of statistics that are gathered: **N** VARCHAR column statistical values are not padded **blank** Statistics have not been collects or VARCHAR column statistical values are padded	G

SYSIBM.SYSKEYTGTDIST table

The SYSIBM.SYSKEYTGTDIST table contains one or more rows for the first key-target of an extended index key.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
STATSTIME	TIMESTAMP NOT NULL WITH DEFAULT	If the RUNSTATS utility updated the statistics, this column contains the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.00.000000'.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
IXSCHEMA	VARCHAR(128) NOT NULL	The qualifier of the index.	G
IXNAME	VARCHAR(128) NOT NULL	The name of the index.	G
KEYSEQ	SMALLINT NOT NULL	The numeric position of the key-target in the index.	G
KEYVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	KEYVALUE contains the data of a frequently occurring value. If the value has a non-character data type, the data might not be printable.	G
TYPE	CHAR(1) NOT NULL WITH DEFAULT 'F'	The type of statistics that are gathered: **C** Cardinality **F** Frequent value **N** Non-padded frequent value **H** Histogram statistics	G
CARDF	FLOAT NOT NULL WITH DEFAULT -1	When TYPE='C', CARDF contains the number of distinct values for the key group. When TYPE='H', CARDF contains the number of distinct values for the key group in a quantile indicated by QUANTILENO.	G
KEYGROUPKEYNO	VARCHAR(254) NOT NULL WITH DEFAULT FOR BIT DATA	KEYGROUPKEYNO contains a value that identifies the set of keys that are associated with the statistics. KEYGROUPKEYNO contains 0 if the statistics are only associated with a single key. If the statistics are associated with more than a single key, KEYGROUPKEYNO contains an array of SMALLINT key numbers with a dimension that is equal to the value in NUMKEYS.	G

Column name	Data type	Description	Use
NUMKEYS	SMALLINT NOT NULL WITH DEFAULT -1	The number of keys that are associated with the statistics.	G
FREQUENCYF	FLOAT NOT NULL WITH DEFAULT -1	When TYPE='F' or 'N', FREQUENCYF contains a value that indicates the percentage of entries in the index that have the value that is contained in the KEYVALUE column. When TYPE='H', FREQUENCYF contains a value that indicates the percentage of entries in the index that have a value that is in the range of the quantile that is indicated in the QUALTILENO column. To determine the percentage from the value of FREQUENCYF, multiply the value by 100. For example, a value of '1' indicates 100 percent. A value of '.153' indicates '15.3' percent.	G
QUANTILENO	SMALLINT NOT NULL WITH DEFAULT -1	QUANTILENO contains an ordinary sequence number of a quantile in the whole consecutive value range, from low to high.	G
LOWVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	When TYPE='H', LOWVALUE contains the lower bound for the quantile that is in QUANTILENO. LOWVALUE is not used if TYPE does not equal 'H'.	G
HIGHVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	When TYPE='H', HIGHVALUE contains the upper bound for the quantile that is in QUANTILENO. HIGHVALUE is not used if TYPE does not equal 'H'.	G

SYSIBM.SYSKEYTGTDISTSTATS table

The SYSIBM.SYSKEYTGTDISTSTATS table contains zero or more rows per partition for the first key-target of a data-partitioned secondary index.

Rows are inserted when RUNSTATS scans a data-partitioned secondary index. No row is inserted if the index is a secondary index. Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
STATSTIME	TIMESTAMP NOT NULL WITH DEFAULT	If RUNSTATS updated the statistics, STATSTIME contains the timestamp of the most recent RUNSTATS. The default value is '0001-01-01.00.00.00.000000'.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
PARTITION	SMALLINT NOT NULL	The partition number of the table space that contains the index in which the key is defined.	G
IXSCHEMA	VARCHAR(128) NOT NULL	The qualifier of the index.	G
IXNAME	VARCHAR(128) NOT NULL	The name of the index.	G
KEYSEQ	SMALLINT NOT NULL	Numeric position of the key-target in the index.	G
KEYVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	KEYVALUE contains the data of a frequently occurring value. If the value has a non-character data type, the data might not be printable.	G
TYPE	CHAR(1) NOT NULL WITH DEFAULT 'F'	The type of statistics that are gathered: **C** Cardinality **F** Frequent value **N** Non-padded frequent value **H** Histogram statistics	G
CARDF	FLOAT NOT NULL WITH DEFAULT -1	When TYPE='C', CARDF contains the number of distinct values for the key group. When TYPE='H', CARDF contains the number of distinct values for the key group in the quantile that is in QUANTILENO.	G
KEYGROUPKEYNO	VARCHAR(254) NOT NULL WITH DEFAULT	Identifies the set of keys that are associated with the statistics. If the statistics are only associated with a single key, KEYGROUPKEYNO contains a zero length value. Otherwise, KEYGROUPKEYNO contains an array of SMALLINT key numbers that have a dimension that is equal to the value in NUMKEYS.	G

Column name	Data type	Description	Use
NUMKEYS	SMALLINT NOT NULL WITH DEFAULT	Identifies the number of keys that are associated with the statistics.	G
FREQUENCYF	FLOAT NOT NULL WITH DEFAULT -1	When TYPE='F' or 'N', FREQUENCYF contains the percentage of entries in the index that have the value that is specified in KEYVALUE when the number of entries is multiplied by 100. For example, a value of '1' indicates 100 percent. A value of '.153' indicates 15.3 percent. When TYPE='H', FREQUENCYF contains the percentage of entries in the index that have a value that is in the range of the quantile that is indicated in QUALTILENO.	G
QUANTILENO	SMALLINT NOT NULL WITH DEFAULT -1	QUANTILENO contains an ordinary sequence number of a quantile in the whole consecutive value range, from low to high.	G
LOWVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	When TYPE='H', LOWVALUE is the lower bound for the quantile that is indicated in QUANTILENO. LOWVALUE is not used if TYPE does not equal 'H'.	G
HIGHVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	When TYPE='H', HIGHVALUE is the upper bound for the quantile that is indicated in QUANTILENO. HIGHVALUE is not used if TYPE does not equal 'H'.	G
	VARCHAR(1000)	Internal use only	I

SYSIBM.SYSKEYTGTDIST_HIST table

The SYSIBM.SYSKEYTGTDIST_HIST table contains rows from the SYSKEYTGTDIST table. Whenever rows are added or changed in SYSKEYTGTDIST, the rows are also written to this table.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
STATSTIME	TIMESTAMP NOT NULL WITH DEFAULT	If the RUNSTATS utility updated the statistics, this column contains the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.00.000000'.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
IXSCHEMA	VARCHAR(128) NOT NULL	The qualifier of the index.	G
IXNAME	VARCHAR(128) NOT NULL	The name of the index.	G
KEYSEQ	SMALLINT NOT NULL	The numeric position of the key-target in the index.	G
KEYVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	KEYVALUE contains the data of a frequently occurring value. If the value has a non-character data type, the data might not be printable.	G
TYPE	CHAR(1) NOT NULL WITH DEFAULT 'F'	The type of statistics that are gathered: **C** Cardinality **F** Frequent value **N** Non-padded frequent value **H** Histogram statistics	G
CARDF	FLOAT NOT NULL WITH DEFAULT -1	When TYPE='C', CARDF contains the number of distinct values for the key group. When TYPE='H', CARDF contains the number of distinct values for the key group in a quantile indicated by QUANTILENO.	G
KEYGROUPKEYNO	VARCHAR(254) NOT NULL WITH DEFAULT FOR BIT DATA	KEYGROUPKEYNO contains a value that identifies the set of keys that are associated with the statistics. KEYGROUPKEYNO contains 0 if the statistics are only associated with a single key. If the statistics are associated with more than a single key, KEYGROUPKEYNO contains an array of SMALLINT key numbers with a dimension that is equal to the value in NUMKEYS.	G

Column name	Data type	Description	Use
NUMKEYS	SMALLINT NOT NULL WITH DEFAULT -1	The number of keys that are associated with the statistics.	G
FREQUENCYF	FLOAT NOT NULL WITH DEFAULT -1	When TYPE='F' or 'N', FREQUENCYF contains the percentage of entries in the index that have the value that is specified in KEYVALUE when the number of entries is multiplied by 100. For example, a value of '1' indicates 100 percent. A value of '.153' indicates 15.3 percent. When TYPE='H', FREQUENCYF contains the percentage of entries in the index that have a value that is in the range of the quantile that is indicated in QUALTILENO.	G
QUANTILENO	SMALLINT NOT NULL WITH DEFAULT -1	QUANTILENO contains an ordinary sequence number of a quantile in the whole consecutive value range, from low to high.	G
LOWVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	When TYPE='H', LOWVALUE contains the lower bound for the quantile that is in QUANTILENO. LOWVALUE is not used if TYPE does not equal 'H'.	G
HIGHVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	When TYPE='H', HIGHVALUE contains the upper bound for the quantile that is in QUANTILENO. HIGHVALUE is not used if TYPE does not equal 'H'.	G

SYSIBM.SYSLOBSTATS table

The SYSIBM.SYSLOBSTATS table contains one row for each LOB table space.

Column name	Data type	Description	Use
STATSTIME	TIMESTAMP NOT NULL	Timestamp of RUNSTATS statistics update.	G
AVGSIZE	INTEGER NOT NULL	Average size of a LOB, measured in bytes, in the LOB table space.	S
FREESPACE	INTEGER NOT NULL	Number of kilobytes of available space in the LOB table space.	S
ORGRATIO	DECIMAL(5,2) NOT NULL	The percentage of organization in the LOB table space. A value of '100' indicates perfect organization of the LOB table space. A value of '1' indicates that the LOB table space is disorganized. A value of '0' indicates that the LOB table space is totally disorganized.	S
DBNAME	VARCHAR(24) NOT NULL	Name of the database that contains the LOB table space named in NAME.	G
NAME	VARCHAR(24) NOT NULL	Name of the LOB table space.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

SYSIBM.SYSLOBSTATS_HIST table

The SYSIBM.SYSLOBSTATS_HIST table contains rows from SYSIBM.SYSLOBSTATS.

Rows are added or changed in this table when RUNSTATS collects history statistics. Rows in this table can also be inserted, updated, and deleted.

Column name	Data type	Description	Use
STATSTIME	TIMESTAMP NOT NULL	Timestamp of RUNSTATS statistics update.	G
FREESPACE	INTEGER NOT NULL	Number of pages of free space in the LOB table space.	S
ORGRATIO	DECIMAL(5,2) NOT NULL	The percentage of organization in the LOB table space. A value of '100' indicates perfect organization of the LOB table space. A value of '1' indicates that the LOB table space is disorganized. A value of '0' indicates that the LOB table space is totally disorganized.	S
DBNAME	VARCHAR(24) NOT NULL	Name of the database that contains the LOB table space named in NAME.	G
NAME	VARCHAR(24) NOT NULL	Name of the LOB table space.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

SYSIBM.SYSOBJROLEDEP table

The SYSIBM.SYSOBJROLEDEP table lists the dependent objects for each role.

Column name	Data type	Description	Use
DEFINER	VARCHAR(128) NOT NULL	The authorization ID or role that created the object.	G
DEFINERTYPE	CHAR(1) NOT NULL	The type of definer: **L** Role **blank** Authorization ID	G
ROLENAME	VARCHAR(128) NOT NULL	Name of the role on which there is a dependency.	G
DSCHEMA	VARCHAR(128) NOT NULL	Name of the schema of the dependent object.	G
DNAME	VARCHAR(762) NOT NULL	Name of the dependent object.	G
DTYPE	CHAR(1) NOT NULL	The type of the dependent object in DNAME: **B** Trigger **D** Database **E** Distinct type **F** User-defined function **I** Index **J** JAR file **L** Role **M** Materialized query table **N** Trusted context **O** Stored procedure **Q** Sequence **R** Table space **S** Storage group **T** Table **V** View **X** Row permission **Y** Column mask **0 (zero)** Alias	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

SYSIBM.SYSPACKAGE table

The SYSIBM.SYSPACKAGE table contains a row for every package.

Column name	Data type	Description	Use
LOCATION	VARCHAR(128) NOT NULL	Always contains blanks	S
COLLID	VARCHAR(128) NOT NULL	Name of the package collection. For a trigger package, it is the schema name of the trigger.	G
NAME	VARCHAR(128) NOT NULL	Name of the package.	G
CONTOKEN	CHAR(8) NOT NULL FOR BIT DATA	Consistency token for the package. For a package derived from a DB2 DBRM, this is either: • The "level" as specified by the LEVEL option when the package's program was precompiled • The timestamp indicating when the package's program was precompiled, in an internal format.	S
OWNER	VARCHAR(128) NOT NULL	Authorization ID of the package owner. For a trigger package, the value is the authorization ID of the owner of the trigger, which is set to the current authorization ID (the plan or package owner for static CREATE TRIGGER statement; the CURRENT SQLID for a dynamic CREATE TRIGGER statement).	G
CREATOR	VARCHAR(128) NOT NULL	Authorization ID of the owner of the creator of the package version. For a trigger package, the value is determined differently. For dynamic SQL, it is the primary authorization ID of the user who issued the CREATE TRIGGER statement. For static SQL, it is the authorization ID of the plan or package owner.	G
TIMESTAMP	TIMESTAMP NOT NULL	Timestamp indicating when the package was created.	G
BINDTIME	TIMESTAMP NOT NULL	Timestamp indicating when the package was last bound.	G
QUALIFIER	VARCHAR(128) NOT NULL	Implicit qualifier for the unqualified table, view, index, and alias names in the static SQL statements of the package.	G
PKSIZE	INTEGER NOT NULL	Size of the base section[40] of the package, in bytes.	G
AVGSIZE	INTEGER NOT NULL	Average size, in bytes, of those sections[40] of the plan that contain SQL statements processed at bind time.	G
SYSENTRIES	SMALLINT NOT NULL	Number of enabled or disabled entries for this package in SYSIBM.SYSPKSYSTEM. A value of 0 if all types of connections are enabled.	G

40. Packages are divided into *sections*. The base section of the package must be in the EDM pool during the entire time the package is executing. Other sections of the package, corresponding roughly to sets of related SQL statements, are brought into the pool as needed.

Column name	Data type	Description	Use
VALID	CHAR(1) NOT NULL	Whether the package is valid: **A** An ALTER statement changed the description of the table or base table of a view referred to by the package. For a CREATE INDEX statement involving data sharing, VALID is also marked as "A". The changes do not invalidate the package. **H** An ALTER TABLE statement changed the description of the table or base table of a view referred to by the package. For releases of DB2 prior to Version 5, the change invalidates the package. **N** No **Y** Yes	G
OPERATIVE	CHAR(1) NOT NULL	Whether the package can be allocated: **N** No; an explicit BIND or REBIND is required before the package can be allocated. **Y** Yes	G
VALIDATE	CHAR(1) NOT NULL	Whether validity checking can be deferred until run time: **B** All checking must be performed at bind time. **R** Validation is done at run time for tables, views, and privileges that do not exist at bind time.	G
ISOLATION	CHAR(1) NOT NULL	Isolation level when the package was last bound or rebound **R** RR (repeatable read) **S** CS (cursor stability) **T** RS (read stability) **U** UR (uncommitted read) **blank** Not specified, and therefore at the level specified for the plan executing the package	G
RELEASE	CHAR(1) NOT NULL	The value used for RELEASE when the package was last bound or rebound: **C** Value used was COMMIT. **D** Value used was DEALLOCATE. **I** The local package is inheriting the value from the plan **blank** Not specified, and therefore the value specified for the plan executing the package.	G
EXPLAIN	CHAR(1) NOT NULL	EXPLAIN option specified for the package; that is, whether information on the package's statements was added to the owner of the PLAN_TABLE table: **N** No **Y** Yes	G
QUOTE	CHAR(1) NOT NULL	SQL string delimiter for SQL statements in the package: **N** Apostrophe **Y** Quotation mark	G
COMMA	CHAR(1) NOT NULL	Decimal point representation for SQL statements in package: **N** Period **Y** Comma	G

Column name	Data type	Description	Use
HOSTLANG	CHAR(1) NOT NULL	Host language for the package's DBRM: **B** Assembler language **C** OS/VS COBOL **D** C **F** Fortran **P** PL/I **2** VS COBOL II or IBM COBOL Release 1 (formerly called COBOL/370) **3** IBM COBOL (Release 2 or subsequent releases) **4** C++ **blank** For remotely bound packages, trigger packages (TYPE='T'), SQL procedure packages (TYPE='N'), or non-inline SQL scalar function packages (TYPE='F').	G
CHARSET	CHAR(1) NOT NULL	Indicates whether the system CCSID for SBCS data was 290 (Katakana) when the program was precompiled: **K** Yes **A** No	G
MIXED	CHAR(1) NOT NULL	Indicates if mixed data was in effect when the package's program was precompiled (for more on when mixed data is in effect, see “Character strings” on page 84): **N** No **Y** Yes	G
DEC31	CHAR(1) NOT NULL	Indicates whether DEC31 was in effect when the package's program was precompiled (for more on when DEC31 is in effect, see “Arithmetic with two decimal operands” on page 244): **N** No **Y** Yes	G
DEFERPREP	CHAR(1) NOT NULL	Indicates the CURRENTDATA option when the package was bound or rebound: **A** Data currency is required for all cursors. Inhibit blocking for all cursors. **B** Data currency is not required for ambiguous cursors. **C** Data currency is required for ambiguous cursors. **blank** The package was created before the CURRENTDATA option was available.	G
SQLERROR	CHAR(1) NOT NULL	Indicates the SQLERROR option on the most recent subcommand that bound or rebound the package: **C** CONTINUE **N** NOPACKAGE	G
REMOTE	CHAR(1) NOT NULL	Source of the package: **C** Package was created by BIND COPY. **D** Package was created by BIND COPY with the OPTIONS(COMMAND) option. **K** The package was copied from a package that was originally bound on behalf of a remote requester. **L** The package was copied with the OPTIONS(COMMAND) option from a package that was originally bound on behalf of a remote requester. **N** Package was locally bound from a DBRM. **Y** Package was bound on behalf of a remote requester.	G

Column name	Data type	Description	Use
PCTIMESTAMP	TIMESTAMP NOT NULL	Date and time the application program was precompiled, or '0001-01-01-00.00.00.000000' if the LEVEL precompiler option was used, or if the package came from a non-DB2 location.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies. RELBOUND should be used instead.	G
VERSION	VARCHAR(122) NOT NULL	Version identifier for the package. The value is blank for a trigger package (TYPE='T') and when the package is created using the BIND PACKAGE command (the initial version of the package)(TYPE='blank').	G
PDSNAME	VARCHAR(132) NOT NULL	For a locally bound package, the name of the PDS (library) in which the package's DBRM is a member. For a locally copied package, the value in SYSPACKAGE.PDSNAME for the source package. Otherwise, the product signature of the bind requester followed by one of the following: • For DB2 for z/OS remote requesters, the requester's location name, or IP address, or LU name enclosed in angle brackets (for example, "<LUSQLDS>"). • For non-DB2 for z/OS remote requesters, the requester's IP address or LU name enclosed in angle brackets.	G
DEGREE	CHAR(3) NOT NULL WITH DEFAULT	The DEGREE option used when the package was last bound: **ANY** DEGREE(ANY) **1 or blank** DEGREE(1). Blank if the package was migrated.	G
GROUP_MEMBER	VARCHAR(24) NOT NULL WITH DEFAULT	The DB2 data sharing member name of the DB2 subsystem that performed the most recent bind. This column is blank if the DB2 subsystem was not in a DB2 data sharing environment when the bind was performed.	G

Column name	Data type	Description	Use
DYNAMICRULES	CHAR(1) NOT NULL WITH DEFAULT	The DYNAMICRULES option used when the package was last bound: **B** BIND. Dynamic SQL statements are executed with DYNAMICRULES bind behavior. **D** DEFINEBIND. When the package is run under an active stored procedure or user-defined function, dynamic SQL statements in the package are executed with DYNAMICRULES define behavior. When the package is not run under an active stored procedure or user-defined function, dynamic SQL statements in the package are executed with DYNAMICRULES bind behavior. **E** DEFINERUN. When the package is run under an active stored procedure or user-defined function, dynamic SQL statements in the package are executed with DYNAMICRULES define behavior. When the package is not run under an active stored procedure or user-defined function, dynamic SQL statements in the package are executed with DYNAMICRULES run behavior. **H** INVOKEBIND. When the package is run under an active stored procedure or user-defined function, dynamic SQL statements in the package are executed with DYNAMICRULES invoke behavior. When the package is not run under an active stored procedure or user-defined function, dynamic SQL statements in the package are executed with DYNAMICRULES bind behavior. **I** INVOKERUN. When the package is run under an active stored procedure or user-defined function, dynamic SQL statements in the package are executed with DYNAMICRULES invoke behavior. When the package is not run under an active stored procedure or user-defined function, dynamic SQL statements in the package are executed with DYNAMICRULES run behavior. **R** RUN. Dynamic SQL statements are executed with DYNAMICRULES run behavior. **blank** DYNAMICRULES is not specified for the package. The package uses the DYNAMICRULES value of the plan to which the package is appended at execution time. For a description of the DYNAMICRULES behaviors, see "Authorization IDs and dynamic SQL" on page 75.	G

Column name	Data type	Description	Use
REOPTVAR	CHAR(1) NOT NULL WITH DEFAULT 'N'	Whether the access path is determined again at execution time using input variable values: **A** Bind option REOPT(AUTO) indicates that the access path is determined multiple times at execution time depending on the parameter value. **N** Bind option REOPT(NONE) indicates that the access path is determined at bind time. **Y** Bind option REOPT(ALWAYS) indicates that the access path is determined at execution time for SQL statements with variable values. **1** Bind option REOPT(ONCE) indicates that the access path is determined only once at execution time, using the first set of input variable values, regardless of how many times the same statement is executed.	G
DEFERPREPARE	CHAR(1) NOT NULL WITH DEFAULT	Whether PREPARE processing is deferred until OPEN is executed: **N** Bind option NODEFER(PREPARE) indicates that PREPARE processing is not deferred until OPEN is executed. **Y** Bind option DEFER(PREPARE) indicates that PREPARE processing is deferred until OPEN is executed. **I** The local package is inheriting the value from the plan **blank** Bind option not specified for the package. It is inherited from the plan.	G
KEEPDYNAMIC	CHAR(1) NOT NULL WITH DEFAULT 'N'	Whether prepared dynamic statements are to be purged at each commit point: **N** The bind option is KEEPDYNAMIC(NO). Prepared dynamic SQL statements are destroyed at commit. **Y** The bind option is KEEPDYNAMIC(YES). Prepared dynamic SQL statements are kept past commit.	G
PATHSCHEMAS	VARCHAR(2048) NOT NULL WITH DEFAULT	SQL path specified on the BIND or REBIND command that bound the package. The path is used to resolve unqualified data type, function, and stored procedure names used in certain contexts. If the PATH bind option was not specified, the value in the column is a zero length string; however, DB2 uses the default SQL path.	G

Column name	Data type	Description	Use
TYPE	CHAR(1) NOT NULL WITH DEFAULT	Type of package. Identifies how the package is created: **F** CREATE FUNCTION or ALTER FUNCTION statement, or a BIND PACKAGE DEPLOY command created the package, and this package is a non-inline SQL scalar function package. **N** CREATE PROCEDURE or ALTER PROCEDURE statement, or BIND PACKAGE DEPLOY command created the package, and this package is a native SQL routine package. **R** CREATE TRIGGER or ALTER TRIGGER statement created the package, and the package is a trigger package that has been created or regenerated in Version 11 New Function Mode or later. **T** CREATE TRIGGER statement prior to Version 11 New Function Mode has created the package, and the package is a trigger package. CREATE TRIGGER statement created the package, and the package is a trigger package. **blank** BIND PACKAGE command created the package.	G
DBPROTOCOL	CHAR(1) NOT NULL WITH DEFAULT 'D'	Whether remote access for SQL is implemented with DRDA access or DRDA access with the capability for package-based continuous block fetch: **D** DRDA **C** DRDA access with package-based continuous block fetch enabled.	G
FUNCTIONTS	TIMESTAMP NOT NULL WITH DEFAULT	Timestamp when the function was resolved. Set by the BIND and REBIND commands, but not by AUTOBIND.	G
OPTHINT	VARCHAR(128) NOT NULL WITH DEFAULT	Value of the OPTHINT bind option. Identifies rows in owner.PLAN_TABLE that are to be used as input to DB2. Refer to the ACCESSPATH column in the "SYSIBM.SYSPACKSTMT table" on page 2290 for information about which statements are using the specified hints.	G
ENCODING_CCSID	INTEGER NOT NULL WITH DEFAULT	The CCSID corresponding to the encoding scheme or CCSID as specified for the bind option ENCODING. The Encoding Scheme specified on the bind command: **ccsid** The specified or derived CCSID. **0** The default CCSID as specified on panel DSNTIPF at installation time. Used when the package was bound prior to Version 7.	G
IMMEDWRITE	CHAR(1) NOT NULL WITH DEFAULT	Indicates when writes of updated group buffer pool dependent pages are to be done. This option is only applicable for data sharing environments. **I** The local package is inheriting the value from the plan **N** Bind option IMMEDWRITE(NO) indicates normal write activity is done. **Y** Bind option IMMEDWRITE(YES) indicates that immediate writes are done for updated group buffer pool dependent pages. **1** Bind option IMMEDWRITE(PH1) indicates that updated group buffer pool dependent pages are written at or before phase 1 commit. **blank** A migrated package.	G

Column name	Data type	Description	Use
RELBOUND	CHAR(1) NOT NULL WITH DEFAULT	The release when the package was bound or rebound. **blank** Bound prior to Version 7 For all other values, see Release dependency indicators	G
	CHAR(1)	Not used.	N
REMARKS	VARCHAR(550) NOT NULL WITH DEFAULT	A character string provided by the user with the COMMENT statement.	G
OWNERTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of owner **blank** Authorization ID **L** Role	G
ROUNDING	CHAR(1) NOT NULL WITH DEFAULT	The ROUNDING option used when the package was last bound: **C** ROUND_CEILING **D** ROUND_DOWN **F** ROUND_FLOOR **G** ROUND_HALF_DOWN **E** ROUND_HALF_EVEN **H** ROUND_HALF_UP **U** ROUND_UP **blank** The package created in a DB2 release prior to Version 9.	G
DISTRIBUTE	CHAR(1) NOT NULL WITH DEFAULT 'N'	Determines if DB2 should gather location names from SQL statements, and create remote packages for the user (This only has effect during local bind): **A** DB2 will collect remote location names from SQL statements during local bind, and automatically create remote packages at those sites. The site names are gathered from object names in static SQL statements and literals on CONNECT statements. The sites at which the package is remotely bound can be determined by the location (BTYPE='X') records in SYSIBM.SYSPACKDEP for this package. **L** DB2 will automatically create remote packages at the sites specified in the list of location-names. The sites at which the package is remotely bound can be determined by the location (BTYPE='X') records in SYSIBM.SYSPACKDEP for this package.	G
LASTUSED	DATE NOT NULL WITH DEFAULT	The last date that the corresponding objects are used.	G
CONCUR_ACC_RES	CHAR(1) NOT NULL	Indicates the CONCURRENTACCESSRESOLUTION option when the package was bound or rebound: **blank** Not specified **U** USECURRENTLYCOMMITTED **W** WAITFOROUTCOME	G

Column name	Data type	Description		Use
EXTENDED-INDICATOR	CHAR(1) NOT NULL WITH DEFAULT	The value of the EXTENDEDINDICATOR bind option:		G
		blank	Not specified	
		N	EXTENDEDINDICATOR NO	
		Y	EXTENDEDINDICATOR YES	
	INTEGER NOT NULL	Not used.		N
PLANMGMT	CHAR(1) NOT NULL WITH DEFAULT	The value of the PLANMGMT bind option:		G
		B	PLANMGMT BASIC	
		E	PLANMGMT EXTENDED	
		blank	PLANMGMT OFF	
PLANMGMTSCOPE	CHAR(1) NOT NULL WITH DEFAULT	The value of the PLANMGMTSCOPE bind option:		G
		S	PLANMGMTSCOPE STATIC	
APREUSE	CHAR(1) NOT NULL WITH DEFAULT	The value of the APREUSE bind option:		G
		N	NO or NONE: Access paths are not reused.	
		W	WARN: DB2 tries to reuse access paths. Processing continues when an access path cannot be reused.	
		E	ERROR: DB2 tries to reuse access paths. Processing ends when an access path cannot be reused.	
APRETAINDUP	CHAR(1) NOT NULL WITH DEFAULT	The value of the APRETAINDUP bind option:		G
		Y	APRETAINDUP YES specified. All copies were retained.	
		0	APRETAINDUP NO specified; however, the previous or original package copy is still retained due to access path differences.	
		1	APRETAINDUP NO specified, and the previous package copy is not retained as the access paths are identical to the current copy.	
		2	APRETAINDUP NO specified, and the previous and original package copies are not retained as the access paths are identical to the current copy.	
SYSTIMESENSITIVE	CHAR(1) NOT NULL WITH DEFAULT 'N'	The value of the SYSTIMESENSITIVE bind option:		G
		Y	References to system-period temporal tables are affected by the value of the CURRENT TEMPORAL SYSTEM_TIME special register.	
		N	References to system-period temporal tables are not affected by the value of the CURRENT TEMPORAL SYSTEM_TIME special register.	
	CHAR(1) NOT NULL WITH DEFAULT 'Y'	Not used.		N

Column name	Data type	Description		Use
BUSTIMESENSITIVE	CHAR(1) NOT NULL WITH DEFAULT 'N'	The value of the BUSTIMESENSITIVE bind option:		G
		Y	References to application-period temporal tables are affected by the value of the CURRENT TEMPORAL BUSINESS_TIME special register.	
		N	References to application-period temporal tables are not affected by the value of the CURRENT TEMPORAL BUSINESS_TIME special register.	
APPLCOMPAT	VARCHAR(10) NOT NULL WITH DEFAULT	The value of the APPLCOMPAT bind option:		G
		V10R1	SQL statements in the package have V10R1 compatibility behavior.	
		V11R1	SQL statements in the package have V11R1 compatibility behavior.	
ARCHIVESENSITIVE	CHAR(1) NOT NULL WITH DEFAULT 'N'	The value of the ARCHIVESENSITIVE bind option.		G
		Y	References to archive-enabled tables are affected by the value of the SYSIBMADM.GET_ARCHIVE built-in global variable. Y is the default value.	
		N	References to archive-enabled tables are not affected by the value of the SYSIBMADM.GET_ARCHIVE built-in global variable.	
EXTSEQNO	INTEGER NOT NULL WITH DEFAULT 0	For internal use.		I
DESCSTAT	CHAR(1) NOT NULL WITH DEFAULT	The value of the DESCSTAT bind option.		G
		Y	The DB2 database manager generates a DESCRIBE SQLDA at bind time so that DESCRIBE requests for static SQL can be satisfied during execution.	
		N	The DB2 database manager does not generate a DESCRIBE SQLDA at bind time for static SQL statements.	

SYSIBM.SYSPACKCOPY table

The SYSIBM.SYSPACKCOPY table contains one row for the previous version of each package and one row for the original version of each package.

Column name	Data type	Description	Use
LOCATION	VARCHAR(128) NOT NULL	Always contains blanks	S
COLLID	VARCHAR(128) NOT NULL	Name of the package collection. For a trigger package, it is the schema name of the trigger.	G
NAME	VARCHAR(128) NOT NULL	Name of the package.	G
CONTOKEN	CHAR(8) NOT NULL WITH DEFAULT FOR BIT DATA	Consistency token for the package. For a package derived from a DB2 DBRM, this is either: • The "level" as specified by the LEVEL option when the package's program was precompiled • The timestamp indicating when the package's program was precompiled, in an internal format.	S
OWNER	VARCHAR(128) NOT NULL	Authorization ID of the package owner. For a trigger package, the value is the authorization ID of the owner of the trigger, which is set to the current authorization ID (the plan or package owner for static CREATE TRIGGER statement; the CURRENT SQLID for a dynamic CREATE TRIGGER statement).	G
CREATOR	VARCHAR(128) NOT NULL	Authorization ID of the owner of the creator of the package version. For a trigger package, the value is determined differently. For dynamic SQL, it is the primary authorization ID of the user who issued the CREATE TRIGGER statement. For static SQL, it is the authorization ID of the plan or package owner.	G
TIMESTAMP	TIMESTAMP NOT NULL	Timestamp indicating when the package was created.	G
BINDTIME	TIMESTAMP NOT NULL	Timestamp indicating when the package was last bound.	G
QUALIFIER	VARCHAR(128) NOT NULL	Implicit qualifier for the unqualified table, view, index, and alias names in the static SQL statements of the package.	G
PKSIZE	INTEGER NOT NULL	Size of the base section[41] of the package, in bytes.	G
AVGSIZE	INTEGER NOT NULL	Average size, in bytes, of those sections[41] of the plan that contain SQL statements processed at bind time.	G

41. Packages are divided into *sections*. The base section of the package must be in the EDM pool during the entire time the package is executing. Other sections of the package, corresponding roughly to sets of related SQL statements, are brought into the pool as needed.

Column name	Data type	Description	Use
SYSENTRIES	SMALLINT NOT NULL	Number of enabled or disabled entries for this package in SYSIBM.SYSPKSYSTEM. A value of 0 if all types of connections are enabled.	G
VALID	CHAR(1) NOT NULL	Whether the package is valid: **A** An ALTER statement changed the description of the table or base table of a view referred to by the package. For a CREATE INDEX statement involving data sharing, VALID is also marked as "A". The changes do not invalidate the package. **H** An ALTER TABLE statement changed the description of the table or base table of a view referred to by the package. For releases of DB2 prior to Version 5, the change invalidates the package. **N** No **Y** Yes	G
OPERATIVE	CHAR(1) NOT NULL	Whether the package can be allocated: **N** No; an explicit BIND or REBIND is required before the package can be allocated. **Y** Yes	G
VALIDATE	CHAR(1) NOT NULL	Whether validity checking can be deferred until run time: **B** All checking must be performed at bind time. **R** Validation is done at run time for tables, views, and privileges that do not exist at bind time.	G
ISOLATION	CHAR(1) NOT NULL	Isolation level when the package was last bound or rebound **R** RR (repeatable read) **S** CS (cursor stability) **T** RS (read stability) **U** UR (uncommitted read) **blank** Not specified, and therefore at the level specified for the plan executing the package	G
RELEASE	CHAR(1) NOT NULL	The value used for RELEASE when the package was last bound or rebound: **C** Value used was COMMIT. **D** Value used was DEALLOCATE. **blank** Not specified, and therefore the value specified for the plan executing the package.	G
EXPLAIN	CHAR(1) NOT NULL	EXPLAIN option specified for the package; that is, whether information on the package's statements was added to the owner of the PLAN_TABLE table: **N** No **Y** Yes	G
QUOTE	CHAR(1) NOT NULL	SQL string delimiter for SQL statements in the package: **N** Apostrophe **Y** Quotation mark	G
COMMA	CHAR(1) NOT NULL	Decimal point representation for SQL statements in package: **N** Period **Y** Comma	G

Column name	Data type	Description	Use
HOSTLANG	CHAR(1) NOT NULL	Host language for the package's DBRM: **B** Assembler language **C** OS/VS COBOL **D** C **F** Fortran **P** PL/I **2** VS COBOL II or IBM COBOL Release 1 (formerly called COBOL/370) **3** IBM COBOL (Release 2 or subsequent releases) **4** C++ **blank** For remotely bound packages, trigger packages (TYPE='T'), SQL procedure packages (TYPE='N'), or non-inline SQL scalar function packages (TYPE='F').	G
CHARSET	CHAR(1) NOT NULL	Indicates whether the system CCSID for SBCS data was 290 (Katakana) when the program was precompiled: **K** Yes **A** No	G
MIXED	CHAR(1) NOT NULL	Indicates if mixed data was in effect when the package's program was precompiled (for more on when mixed data is in effect, see “Character strings” on page 84): **N** No **Y** Yes	G
DEC31	CHAR(1) NOT NULL	Indicates whether DEC31 was in effect when the package's program was precompiled (for more on when DEC31 is in effect, see “Arithmetic with two decimal operands” on page 244): **N** No **Y** Yes	G
DEFERPREP	CHAR(1) NOT NULL	Indicates the CURRENTDATA option when the package was bound or rebound: **A** Data currency is required for all cursors. Inhibit blocking for all cursors. **B** Data currency is not required for ambiguous cursors. **C** Data currency is required for ambiguous cursors. **blank** The package was created before the CURRENTDATA option was available.	G
SQLERROR	CHAR(1) NOT NULL	Indicates the SQLERROR option on the most recent subcommand that bound or rebound the package: **C** CONTINUE **N** NOPACKAGE	G
REMOTE	CHAR(1) NOT NULL	Source of the package: **C** Package was created by BIND COPY. **D** Package was created by BIND COPY with the OPTIONS(COMMAND) option. **K** The package was copied from a package that was originally bound on behalf of a remote requester. **L** The package was copied with the OPTIONS(COMMAND) option from a package that was originally bound on behalf of a remote requester. **N** Package was locally bound from a DBRM. **Y** Package was bound on behalf of a remote requester.	G

Column name	Data type	Description	Use
PCTIMESTAMP	TIMESTAMP NOT NULL	Date and time the application program was precompiled, or '0001-01-01-00.00.00.000000' if the LEVEL precompiler option was used, or if the package came from a non-DB2 location.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies. RELBOUND should be used instead.	G
VERSION	VARCHAR(122) NOT NULL	Version identifier for the package. The value is blank for a trigger package (TYPE='T').	G
PDSNAME	VARCHAR(132) NOT NULL	For a locally bound package, the name of the PDS (library) in which the package's DBRM is a member. For a locally copied package, the value in SYSPACKAGE.PDSNAME for the source package. Otherwise, the product signature of the bind requester followed by one of the following: • For DB2 for z/OS remote requesters, the requester's location name, or IP address, or LU name enclosed in angle brackets (for example, “<LUSQLDS>”). • For non-DB2 for z/OS remote requesters, the requester's IP address or LU name enclosed in angle brackets.	G
DEGREE	CHAR(3) NOT NULL WITH DEFAULT	The DEGREE option used when the package was last bound: **ANY** DEGREE(ANY) **1 or blank** DEGREE(1). Blank if the package was migrated.	G
GROUP_MEMBER	VARCHAR(24) NOT NULL WITH DEFAULT	The DB2 data sharing member name of the DB2 subsystem that performed the most recent bind. This column is blank if the DB2 subsystem was not in a DB2 data sharing environment when the bind was performed.	G

Column name	Data type	Description	Use
DYNAMICRULES	CHAR(1) NOT NULL WITH DEFAULT	The DYNAMICRULES option used when the package was last bound: **B** BIND. Dynamic SQL statements are executed with DYNAMICRULES bind behavior. **D** DEFINEBIND. When the package is run under an active stored procedure or user-defined function, dynamic SQL statements in the package are executed with DYNAMICRULES define behavior. When the package is not run under an active stored procedure or user-defined function, dynamic SQL statements in the package are executed with DYNAMICRULES bind behavior. **E** DEFINERUN. When the package is run under an active stored procedure or user-defined function, dynamic SQL statements in the package are executed with DYNAMICRULES define behavior. When the package is not run under an active stored procedure or user-defined function, dynamic SQL statements in the package are executed with DYNAMICRULES run behavior. **H** INVOKEBIND. When the package is run under an active stored procedure or user-defined function, dynamic SQL statements in the package are executed with DYNAMICRULES invoke behavior. When the package is not run under an active stored procedure or user-defined function, dynamic SQL statements in the package are executed with DYNAMICRULES bind behavior. **I** INVOKERUN. When the package is run under an active stored procedure or user-defined function, dynamic SQL statements in the package are executed with DYNAMICRULES invoke behavior. When the package is not run under an active stored procedure or user-defined function, dynamic SQL statements in the package are executed with DYNAMICRULES run behavior. **R** RUN. Dynamic SQL statements are executed with DYNAMICRULES run behavior. **blank** DYNAMICRULES is not specified for the package. The package uses the DYNAMICRULES value of the plan to which the package is appended at execution time. For a description of the DYNAMICRULES behaviors, see "Authorization IDs and dynamic SQL" on page 75.	G

Column name	Data type	Description	Use
REOPTVAR	CHAR(1) NOT NULL WITH DEFAULT 'N'	Whether the access path is determined again at execution time using input variable values: **A** Bind option REOPT(AUTO) indicates that the access path is determined multiple times at execution time depending on the parameter value. **N** Bind option REOPT(NONE) indicates that the access path is determined at bind time. **Y** Bind option REOPT(ALWAYS) indicates that the access path is determined at execution time for SQL statements with variable values. **1** Bind option REOPT(ONCE) indicates that the access path is determined only once at execution time, using the first set of input variable values, regardless of how many times the same statement is executed.	G
DEFERPREPARE	CHAR(1) NOT NULL WITH DEFAULT	Whether PREPARE processing is deferred until OPEN is executed: **N** Bind option NODEFER(PREPARE) indicates that PREPARE processing is not deferred until OPEN is executed. **Y** Bind option DEFER(PREPARE) indicates that PREPARE processing is deferred until OPEN is executed. **blank** Bind option not specified for the package. It is inherited from the plan.	G
KEEPDYNAMIC	CHAR(1) NOT NULL WITH DEFAULT 'N'	Whether prepared dynamic statements are to be purged at each commit point: **N** The bind option is KEEPDYNAMIC(NO). Prepared dynamic SQL statements are destroyed at commit. **Y** The bind option is KEEPDYNAMIC(YES). Prepared dynamic SQL statements are kept past commit.	G
PATHSCHEMAS	VARCHAR(2048) NOT NULL WITH DEFAULT	SQL path specified on the BIND or REBIND command that bound the package. The path is used to resolve unqualified data type, function, and stored procedure names used in certain contexts. If the PATH bind option was not specified, the value in the column is a zero length string; however, DB2 uses the default SQL path.	G

Column name	Data type	Description	Use
TYPE	CHAR(1) NOT NULL WITH DEFAULT	Type of package. Identifies how the package is created: **F** CREATE FUNCTION or ALTER FUNCTION statement, or a BIND PACKAGE DEPLOY command created the package, and this package is a non-inline SQL scalar function package. **N** CREATE PROCEDURE or ALTER PROCEDURE statement, or BIND PACKAGE DEPLOY command created the package, and this package is a native SQL routine package. **R** CREATE TRIGGER or ALTER TRIGGER statement created the package, and the package is a trigger package that has been created or regenerated in Version 11 New Function Mode or later. **T** CREATE TRIGGER statement prior to Version 11 New Function Mode has created the package, and the package is a trigger package. CREATE TRIGGER statement created the package, and the package is a trigger package. **blank** BIND PACKAGE command created the package.	G
DBPROTOCOL	CHAR(1) NOT NULL WITH DEFAULT 'P'	Whether remote access for SQL is implemented with DRDA access or DRDA access with the capability for package-based continuous block fetch: **D** DRDA **C** DRDA access with the capability for package-based continuous block fetch.	G
FUNCTIONTS	TIMESTAMP NOT NULL WITH DEFAULT	Timestamp when the function was resolved. Set by the BIND and REBIND commands, but not by AUTOBIND.	G
OPTHINT	VARCHAR(128) NOT NULL WITH DEFAULT	Value of the OPTHINT bind option. Identifies rows in owner.PLAN_TABLE that are to be used as input to DB2. Refer to the ACCESSPATH column in the "SYSIBM.SYSPACKSTMT table" on page 2290 for information about which statements are using the specified hints.	G
ENCODING_CCSID	INTEGER NOT NULL WITH DEFAULT	The CCSID corresponding to the encoding scheme or CCSID as specified for the bind option ENCODING. The Encoding Scheme specified on the bind command: **ccsid** The specified or derived CCSID. **0** The default CCSID as specified on panel DSNTIPF at installation time. Used when the package was bound prior to Version 7.	G
IMMEDWRITE	CHAR(1) NOT NULL WITH DEFAULT	Indicates when writes of updated group buffer pool dependent pages are to be done. This option is only applicable for data sharing environments. **N** Bind option IMMEDWRITE(NO) indicates normal write activity is done. **Y** Bind option IMMEDWRITE(YES) indicates that immediate writes are done for updated group buffer pool dependent pages. **1** Bind option IMMEDWRITE(PH1) indicates that updated group buffer pool dependent pages are written at or before phase 1 commit. **blank** A migrated package.	G

Column name	Data type	Description	Use
RELBOUND	CHAR(1) NOT NULL WITH DEFAULT	The release when the package was bound or rebound. **blank** Bound prior to Version 7 For all other values, see Release dependency indicators	G
	CHAR(1)	Not used.	N
REMARKS	VARCHAR(550) NOT NULL WITH DEFAULT	A character string provided by the user with the COMMENT statement.	G
OWNERTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of owner **blank** Authorization ID **L** Role	G
ROUNDING	CHAR(1) NOT NULL WITH DEFAULT	The ROUNDING option used when the package was last bound: **C** ROUND_CEILING **D** ROUND_DOWN **F** ROUND_FLOOR **G** ROUND_HALF_DOWN **E** ROUND_HALF_EVEN **H** ROUND_HALF_UP **U** ROUND_UP **blank** The package created in a DB2 release prior to Version 9.	G
DISTRIBUTE	CHAR(1) NOT NULL WITH DEFAULT 'N'	Determines if DB2 should gather location names from SQL statements, and create remote packages for the user (This only has effect during local bind): **A** DB2 will collect remote location names from SQL statements during local bind, and automatically create remote packages at those sites. The site names are gathered from object names in static SQL statements and literals on CONNECT statements. The sites at which the package is remotely bound can be determined by the location (BTYPE='X') records in SYSIBM.SYSPACKDEP for this package. **L** DB2 will automatically create remote packages at the sites specified in the list of location-names. The sites at which the package is remotely bound can be determined by the location (BTYPE='X') records in SYSIBM.SYSPACKDEP for this package.	G
LASTUSED	DATE NOT NULL WITH DEFAULT	The last date that the corresponding objects are used.	G
CONCUR_ACC_RES	CHAR(1) NOT NULL'	Indicates the CONCURRENTACCESSRESOLUTION option when the package was bound or rebound: **blank** Not specified **U** USECURRENTLYCOMMITTED **W** WAITFOROUTCOME	G

Column name	Data type	Description		Use
EXTENDED-INDICATOR	CHAR(1) NOT NULL WITH DEFAULT	The value of the EXTENDEDINDICATOR bind option:		G
		N	EXTENDEDINDICATOR NO	
		Y	EXTENDEDINDICATOR YES	
COPYID	INTEGER NOT NULL	The version of the copy of the package that this row explains:		G
		1	The previous copy of the package	
		2	The original copy of the package	
PLANMGMT	CHAR(1) NOT NULL WITH DEFAULT	The value of the PLANMGMT bind option:		G
		B	PLANMGMT BASIC	
		E	PLANMGMT EXTENDED	
		F	PLANMGMT OFF	
		O	PLANMGMT ON	
PLANMGMTSCOPE	CHAR(1) NOT NULL WITH DEFAULT	The value of the PLANMGMTSCOPE bind option:		G
		S	PLANMGMTSCOPE STATIC	
APREUSE	CHAR(1) NOT NULL WITH DEFAULT	The value of the APREUSE bind option:		I
		N	NO or NONE: Access paths are not reused.	
		E	ERROR: DB2 tries to reuse access paths. Processing ends when an access path cannot be reused.	
APRETAINDUP	CHAR(1) NOT NULL WITH DEFAULT	The value of the APRETAINDUP bind option:		G
		Y	APRETAINDUP YES specified. All copies were retained.	
		0	APRETAINDUP NO specified; however, the previous or original package copy is still retained due to access path differences.	
		1	APRETAINDUP NO specified, and the previous package copy is not retained as the access paths are identical to the current copy.	
		2	APRETAINDUP NO specified, and the previous and original package copies are not retained as the access paths are identical to the current copy.	
SYSTIMESENSITIVE	CHAR(1) NOT NULL WITH DEFAULT 'N'	The value of the SYSTIMESENSITIVE bind option:		G
		Y	References to system-period temporal tables are affected by the value of the CURRENT TEMPORAL SYSTEM_TIME special register.	
		N	References to system-period temporal tables are not affected by the value of the CURRENT TEMPORAL SYSTEM_TIME special register.	
	CHAR(1) NOT NULL WITH DEFAULT 'Y'	Not used.		N

Column name	Data type	Description	Use
BUSTIMESENSITIVE	CHAR(1) NOT NULL WITH DEFAULT 'N'	The value of the BUSTIMESENSITIVE bind option: **Y** References to application-period temporal tables are affected by the value of the CURRENT TEMPORAL BUSINESS_TIME special register. **N** References to application-period temporal tables are not affected by the value of the CURRENT TEMPORAL BUSINESS_TIME special register.	G
APPLCOMPAT	VARCHAR(10) NOT NULL WITH DEFAULT 'Y'	The value of the APPLCOMPAT bind option: **V10R1** SQL statements in the package have V10R1 compatibility behavior. **V11R1** SQL statements in the package have V11R1 compatibility behavior.	G
ARCHIVESENSITIVE	CHAR(1) NOT NULL WITH DEFAULT 'N	The value of the ARCHIVESENSITIVE bind option. **Y** References to archive-enabled tables are affected by the value of the SYSIBMADM.GET_ARCHIVE built-in global variable. Y is the default value. **N** References to archive-enabled tables are not affected by the value of the SYSIBMADM.GET_ARCHIVE built-in global variable.	G
EXTSEQNO	INTEGER NOT NULL WITH DEFAULT 0	For internal use.	I
DESCSTAT	CHAR(1) NOT NULL WITH DEFAULT	The value of the DESCSTAT bind option. **Y** The DB2 database manager generates a DESCRIBE SQLDA at bind time so that DESCRIBE requests for static SQL can be satisfied during execution. **N** The DB2 database manager does not generate a DESCRIBE SQLDA at bind time for static SQL statements.	G

SYSIBM.SYSPACKAUTH table

The SYSIBM.SYSPACKAUTH table records the privileges that are held by users over packages.

Column name	Data type	Description	Use
GRANTOR	VARCHAR(128) NOT NULL	Authorization ID of the user who granted the privilege. Could also be PUBLIC.	G
GRANTEE	VARCHAR(128) NOT NULL	Authorization ID of the user who holds the privileges, the name of a plan that uses the privileges or PUBLIC for a grant to PUBLIC.	G
LOCATION	VARCHAR(128) NOT NULL	Always contains blanks	S
COLLID	VARCHAR(128) NOT NULL	Collection name for the package or packages on which the privilege was granted.	G
NAME	VARCHAR(128) NOT NULL	Name of the package on which the privileges are held. An asterisk (*) if the privileges are held on all packages in a collection.	G
	CHAR(8) NOT NULL FOR BIT DATA	Not used	N
TIMESTAMP	TIMESTAMP NOT NULL	Timestamp indicating when the privilege was granted.	G
GRANTEETYPE	CHAR(1) NOT NULL	Type of grantee: **blank** An authorization ID **L** Role **P** An application plan	G
AUTHHOWGOT	CHAR(1) NOT NULL	Authorization level of the user from whom the privileges were received. This authorization level is not necessarily the highest authorization level of the grantor. **blank** Not applicable **A** PACKADM (on collection *) **C** DBCTRL **D** DBADM **E** SECADM **G** ACCESSCTRL **L** SYSCTRL **M** DBMAINT **P** PACKADM (on a specific collection) **S** SYSADM **T** DATAACCESS	G
BINDAUTH	CHAR(1) NOT NULL	Whether GRANTEE can use the BIND and REBIND subcommands on the package: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G

42. PUBLIC followed by an asterisk (PUBLIC*) denotes PUBLIC AT ALL LOCATIONS. For the conditions where GRANTOR can be PUBLIC or PUBLIC*, see *DB2 Administration Guide*.

Column name	Data type	Description	Use
COPYAUTH	CHAR(1) NOT NULL	Whether GRANTEE can COPY the package: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G
EXECUTEAUTH	CHAR(1) NOT NULL	Whether GRANTEE can execute the package: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
GRANTORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of grantor: **blank** Authorization ID **L** Role	G

SYSIBM.SYSPACKDEP table

The SYSIBM.SYSPACKDEP table records the dependencies of packages on local tables, views, synonyms, table spaces, indexes, aliases, functions, and stored procedures.

Column name	Data type	Description	Use
BNAME	VARCHAR(128) NOT NULL	The name of an object that a package depends on. If BTYPE is B or C, the value is the name of the table on which the period is defined.	G
BQUALIFIER	VARCHAR(128) NOT NULL	The value of the column depends on the type of object: • If BNAME identifies a table space (BTYPE is R), the value is the name of its database. • If BNAME identifies a table on which a period is defined (BTYPE is B or C), the value is the qualifier of that table. • If BNAME identifies user-defined function, a cast function, a stored procedure, or a sequence (BTYPE is F, O, or Q), the value is the schema name. • If BNAME identifies a role, the value is blank. • Otherwise, the value is the schema of BNAME.	G
BTYPE	CHAR(1) NOT NULL	Type of object identified by BNAME and BQUALIFIER: **B** BUSINESS_TIME **C** SYSTEM_TIME **E** INSTEAD OF trigger **F** User-defined function or cast function **G** Global temporary table **I** Index **M** Materialized query table **O** Stored procedure **P** Partitioned table space if it is defined as LARGE or with the DSSIZE parm **Q** Sequence object **R** Table space **S** Synonym **T** Table **U** Distinct type **V** View **W** SYSTEM_TIME period **Z** BUSINESS_TIME period **0 (zero)** Alias	G
DLOCATION	VARCHAR(128) NOT NULL	Always contains blanks	S
DCOLLID	VARCHAR(128) NOT NULL	Name of the package collection.	G
DNAME	VARCHAR(128) NOT NULL	Name of the package.	G

Column name	Data type	Description	Use
DCONTOKEN	CHAR(8) NOT NULL FOR BIT DATA	Consistency token for the package. This is either: • The "level" as specified by the LEVEL option when the package's program was precompiled • The timestamp indicating when the package's program was precompiled, in an internal format.	S
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
DOWNER	VARCHAR(128) NOT NULL WITH DEFAULT	Owner of the package:	G
DTYPE	CHAR(1) NOT NULL WITH DEFAULT	Type of package: **F** Non-inline SQL scalar function **N** Native SQL routine package **O** Original copy of a package **P** Previous copy of a package **R** Reserved for IBM use **T** Trigger package **blank** Not a trigger package or a native SQL routine package	G
DOWNERTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of owner of the package: **blank** Authorization ID **L** Role	G

SYSIBM.SYSPACKLIST table

The SYSIBM.SYSPACKLIST table contains one or more rows for every local application plan bound with a package list. Each row represents a unique entry in the plan's package list.

Column name	Data type	Description	Use
PLANNAME	VARCHAR(24) NOT NULL	Name of the application plan.	G
SEQNO	SMALLINT NOT NULL	Sequence number of the entry in the package list.	G
LOCATION	VARCHAR(128) NOT NULL	Location of the package. Blank if this is local. An asterisk (*) indicates location to be determined at run time.	G
COLLID	VARCHAR(128) NOT NULL	Collection name for the package. An asterisk (*) indicates that the collection name is determined at run time.	G
NAME	VARCHAR(128) NOT NULL	Name of the package. An asterisk (*) indicates an entire collection.	G
TIMESTAMP	TIMESTAMP NOT NULL	Timestamp indicating when the row was created.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

SYSIBM.SYSPACKSTMT table

The SYSIBM.SYSPACKSTMT table contains one or more rows for each statement in a package.

Column name	Data type	Description	Use
LOCATION	VARCHAR(128) NOT NULL	Always contains blanks	S
COLLID	VARCHAR(128) NOT NULL	Name of the package collection.	G
NAME	VARCHAR(128) NOT NULL	Name of the package.	G
CONTOKEN	CHAR(8) NOT NULL FOR BIT DATA	Consistency token for the package. This is either: • The "level" as specified by the LEVEL option when the package's program was precompiled • The timestamp indicating when the package's program was precompiled, in an internal format	S
SEQNO	INTEGER NOT NULL	Not used.	G
STMTNO	SMALLINT NOT NULL	The statement number of the statement in the source program. A statement number greater than 32767 is stored as zero[43] or as a negative number[44]. If the value is zero, see STMTNOI for the statement number.	G
SECTNO	SMALLINT NOT NULL	The section number of the statement.[44]	G
BINDERROR	CHAR(1) NOT NULL	Whether an SQL error was detected at bind time: **N** No **Y** Yes	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
VERSION	VARCHAR(122) NOT NULL	Version identifier for the package.	G
	VARCHAR(3500) NOT NULL WITH DEFAULT FOR BIT DATA	Internal use only.	I

43. Rows in which the value of SEQNO, STMTNO, and SECTNO are zero are for internal use.

44. To convert a negative STMTNO to a meaningful statement number that corresponds to your precompile output, add 65536 to it. For example, -26472 is equivalent to +39064 (-26472 + 65536).

Column name	Data type	Description	Use
ISOLATION	CHAR(1) NOT NULL WITH DEFAULT	Isolation level for the SQL statement: **R** RR (repeatable read) **T** RS (read stability) **S** CS (cursor stability) **U** UR (uncommitted read) **L** RS isolation, with a *lock-clause* **X** RR isolation, with a *lock-clause* **blank** The WITH clause was not specified on this statement. The isolation level is recorded in SYSPACKAGE.ISOLATION and in SYSPLAN.ISOLATION.	G
STATUS	CHAR(1) NOT NULL WITH DEFAULT	Status of binding the statement: **A** Distributed - statement uses DB2 private protocol access. The statement will be parsed and executed at the server using defaults for input variables during access path selection. **B** Distributed - statement uses DB2 private protocol access. The statement will be parsed and executed at the server using values for input variables during access path selection. **C** Compiled - statement was bound successfully using defaults for input variables during access path selection. **D** Distributed - statement references a remote object using a three-part name. DB2 will implicitly use DRDA access either because the DBPROTOCOL bind option was not specified (defaults to DRDA), or the bind option DBPROTOCOL(DRDA) was explicitly specified. This option allows the use of three-part names with DRDA access but it requires that the package be bound at the target remote site. **E** Explain - statement is an SQL EXPLAIN statement. The explain is done at bind time using defaults for input variables during access path selection. **F** Parsed - statement did not bind successfully and VALIDATE(RUN) was used. The statement will be rebound at execution time using values for input variables during access path selection. **G** Compiled - statement bound successfully, but REOPT is specified. The statement will be rebound at execution time using values for input variables during access path selection. **H** Parsed - statement is either a data definition statement or a statement that did not bind successfully and VALIDATE(RUN) was used. The statement will be rebound at execution time using defaults for input variables during access path selection. Data manipulation statements use defaults for input variables during access path selection. **I** Indefinite - statement is dynamic. The statement will be bound at execution time using defaults for input variables during access path selection.	S

Column name	Data type	Description	Use
STATUS (cont.)		**J** Indefinite - statement is dynamic. The statement will be bound at execution time using values for input variables during access path selection. **K** Control - CALL statement. **L** Bad - the statement has some allowable error. The bind continues but the statement cannot be executed. **M** Parsed - statement references a table that is qualified with SESSION and was not bound because the table reference could be for a declared temporary table that will not be defined until the package or plan is run. The SQL statement will be rebound at execution time using values for input variables during access path selection. **blank** The statement is non-executable, or was bound in a DB2 release prior to Version 5.	
ACCESSPATH	CHAR(1) NOT NULL WITH DEFAULT	For static statements, indicates if the access path for the statement is based on user-specified optimization hints: **H** Optimization hints were used. **A** The access path was reused because of the APREUSE bind option. **blank** One of the following situations: • The access path was determined without the use of hints, and a previous access path was not reused. • No access path is associated with the statement. • The statement is a dynamic SQL statement	G
STMTNOI	INTEGER NOT NULL WITH DEFAULT	If the value of STMTNO is zero, the column contains the statement number of the statement in the source program. If both STMTNO and STMTNOI are zero, the statement number is greater than 32767.	G
SECTNOI	INTEGER NOT NULL WITH DEFAULT	The section number of the statement.	G
EXPLAINABLE	CHAR(1) NOT NULL WITH DEFAULT	Contains one of the following values: **Y** Indicates that the SQL statement can be used with the EXPLAIN function and might have rows describing its access path in the owner.PLAN_TABLE. **N** Indicates that the SQL statement does not have any rows describing its access path in the owner.PLAN_TABLE. **blank** Indicates that the SQL statement was bound prior to Version 7.	G

Column name	Data type	Description	Use
QUERYNO	INTEGER NOT NULL WITH DEFAULT –1	The query number of the SQL statement in the source program. SQL statements bound prior to Version 7 have a default value of –1. Statements bound in Version 7 or later use the value specified on the QUERYNO clause on SELECT, UPDATE, INSERT, DELETE, EXPLAIN, DECLARE CURSOR, or REFRESH TABLE statements. If the QUERYNO clause is not specified, the query number is set to the statement number.	G
ROWID	ROWID NULL GENERATED ALWAYS	ROWID column, created for the lob columns in this table.	G
STMT_ID	BIGINT NOT NULL	A unique statement identifier.	G
STATEMENT	CLOB(2M) NOT NULL WITH DEFAULT	The complete text for the SQL statement that the row represents.	G
	BLOB(2M) NOT NULL WITH DEFAULT	Internal use only.	I

Column name	Data type	Description	Use
EXPANSION_REASON	CHAR(2) NOT NULL	This column applies to only static statements that reference archive tables or temporal tables. For other statements, this column is blank. Indicates the effect of the CURRENT TEMPORAL BUSINESS_TIME special register, the CURRENT TEMPORAL SYSTEM_TIME special register, and the SYSIBMADM.GET_ARCHIVE built-in global variable. These items are controlled by the BUSTIMESENSITIVE, SYSTIMESENSITIVE, and ARCHIVESENSITIVE bind options. If one of these special registers or the global variable is set to Y and the corresponding bind option is set to YES, DB2 implicitly adds certain syntax to the statement. This column indicates whether this implicit query transformation occurred when the package was bound and why. For dynamic statements, this column is blank. For static statements, this column can have one of the following values: **A** The statement was bound with implicit query transformation as a result of the SYSIBMADM.GET_ARCHIVE built-in global variable. **B** The statement was bound with implicit query transformation as a result of the CURRENT TEMPORAL BUSINESS_TIME special register. **S** The statement was bound with implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register. **SB** The statement was bound with implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register and the CURRENT TEMPORAL BUSINESS_TIME special register. **blank** One of the following events occurred: • The statement did not bind successfully and the VALIDATE(RUN) bind option was used. • The statement was bound without implicit query transformation. If this column is not blank, you can see the resulting access path for the transformed statement by using EXPLAIN. **Related information:** "References to built-in global variables" on page 223 "CURRENT TEMPORAL BUSINESS_TIME" on page 194 "CURRENT TEMPORAL SYSTEM_TIME" on page 196 BIND and REBIND options (DB2 Commands)	G

SYSIBM.SYSPACKSTMT_STMB table

The SYSIBM.SYSPACKSTMT_STMB table is an auxiliary table for the STMTBLOB column of the SYSIBM.SYSPACKSTMT table and is required to hold LOB data.

Column name	Data type	Description	Use
	BLOB(2M) NOT NULL WITH DEFAULT	Internal use only.	I

SYSIBM.SYSPACKSTMT_STMT table

The SYSIBM.SYSPACKSTMT_STMT table is an auxiliary table for the STATEMENT column of the SYSIBM.SYSPACKSTMT table and is required to hold LOB data.

Column name	Data type	Description	Use
STATEMENT	CLOB(2M) NOT NULL WITH DEFAULT	The complete text for the SQL statement that the row represents.	G

SYSIBM.SYSPARMS table

The SYSIBM.SYSPARMS table contains a row for each parameter of a routine or multiple rows for table parameters (one for each column of the table).

Column name	Data type	Description	Use
SCHEMA	VARCHAR(128) NOT NULL	Schema of the routine.	G
OWNER	VARCHAR(128) NOT NULL	Owner of the routine.	G
NAME	VARCHAR(128) NOT NULL	Name of the routine.	G
SPECIFICNAME	VARCHAR(128) NOT NULL	Specific name of the routine.	G
ROUTINETYPE	CHAR(1) NOT NULL	Type of routine: **F** User-defined function or cast function **P** Stored procedure	G
CAST_FUNCTION	CHAR(1) NOT NULL	Whether the routine is a cast function: **N** Not a cast function **Y** A cast function The only way to get a value of Y is if a user creates a distinct type when DB2 implicitly generates cast functions for the distinct type.	G
PARMNAME	VARCHAR(128) NOT NULL	Name of the parameter. For a table parameter, the parameter name in the row corresponding to the first column of the table is the parameter name specified on CREATE; an empty string or blanks are stored for the parameter name for the rows corresponding to the remaining columns.	G
ROUTINEID	INTEGER NOT NULL	Internal identifier of the routine.	S

Column name	Data type	Description	Use
ROWTYPE	CHAR(1) NOT NULL	The following values indicate the type of parameter described by this row: **P** Input parameter. **O** Output parameter; not applicable for functions **B** Both an input and an output parameter; not applicable for functions **R** Result before casting; not applicable for stored procedures **C** Result after casting; not applicable for stored procedures **S** Input parameter of the underlying built-in source function. For a sourced function and a given ORDINAL value: • The row with ROWTYPE = P describes the input parameter of the user-defined function (identified by ROUTINEID). • The row with ROWTYPE = S describes the corresponding input parameter of the built-in function that is the underlying source function (identified by the SOURCESCHEMA and SOURCESPECIFIC values). A value of 'X' indicates that the row is not used to describe a particular parameter of the routine. Instead, for a routine that was created prior to Version 9, the row is used to record a CCSID for the encoding scheme specified in a PARAMETER CCSID clause, or a DATATYPEID for the representation of the variable length character string parameters of a LANGUAGE C routine, as specified in a PARAMETER VARCHAR clause. For routines created with Version 8 (new function mode) or later releases, the CCSID is recorded in the PARAMETER_CCSID column of SYSROUTINES. For routines created with Version 9 or later releases, the DATATYPEID information to support PARAMETER VARCHAR is recorded in the PARAMETER_VARCHARFORM column of SYSIBM.SYSROUTINES.	G
ORDINAL	SMALLINT NOT NULL	If ROWTYPE is B, O, P, or S, the value is the ordinal number of the parameter within the routine signature. If ROWTYPE is C or R, the value depends on the type of function: • For a scalar function, the value is 0. • For a table function, the value is the ordinal number of the column of the output table. If ROWTYPE is X, the value is 0.	G
TYPESCHEMA	VARCHAR(128) NOT NULL	Schema of the data type of the parameter.	G
TYPENAME	VARCHAR(128) NOT NULL	Name of the data type of the parameter.	G

Column name	Data type	Description	Use
DATATYPEID	INTEGER NOT NULL	For a built-in data type, the internal ID of the built-in type. For a distinct type, the internal ID of the distinct type. When ROWTYPE is X and ORDINAL is 0, a non-zero DATATYPEID indicates that actual representation, for a LANGUAGE C routine, of any varying length string parameters that appear in the routine's parameter list ot in the RETURNS clause.	S
SOURCETYPEID	INTEGER NOT NULL	For a built-in data type, 0. For a distinct type, the internal ID of the built-in data type upon which the distinct type is based.	S
LOCATOR	CHAR(1) NOT NULL	Indicates whether a locator to a value, instead of the actual value, is to be passed or returned when the routine is called: **N** The actual value is to be passed. **Y** A locator to a value is to be passed	G
TABLE	CHAR(1) NOT NULL	The data type of a column for a table parameter: **N** This is not a table parameter. **Y** This is a table parameter.	G
TABLE_COLNO	SMALLINT NOT NULL	For table parameters, the column number of the table. Otherwise, the value is 0.	G
LENGTH	INTEGER NOT NULL	Length attribute of the parameter or result; If the parameter or result length is determined during function resolution, the length attribute can also be 0. In the case of a decimal parameter or result this is the precision. If the parameter is an array, the value is 0.	G
SCALE	SMALLINT NOT NULL	Scale of the data type of the parameter or number of fractional second digits of timestamp or timestamp with time zone parameter. If it is TIMESTAMP parameter where LENGTH is 10 and SCALE is 0, the number of fractional second digits is 6.	G
SUBTYPE	CHAR(1) NOT NULL	If the data type is a distinct type, the subtype of the distinct type, which is based on the subtype of its source type: **B** The subtype is FOR BIT DATA. **S** The subtype is FOR SBCS DATA. **M** The subtype is FOR MIXED DATA. **blank** The source type is not a character type. If the parameter is an array type, the value is blank.	G
CCSID	INTEGER NOT NULL	CCSID of the data type for a character, date, time, timestamp or graphic data type. If the parameter is a datetime array, the value is 0. (not null) When ROWTYPE is X and ORDINAL is 0, the CCSID column is the CCSID for all character and graphic string parameters.	G
CAST_FUNCTION_ID	INTEGER NOT NULL	Internal function ID of the function used to cast the argument, if this function is sourced on another function, or result. Otherwise, the value is 0. Not applicable for stored procedures.	S

Column name	Data type	Description	Use
ENCODING_SCHEME	CHAR(1) NOT NULL	Encoding scheme of the parameter: **A** ASCII **E** EBCDIC **U** UNICODE **blank** The source type is not a character, graphic, or datetime type. If the parameter is an array type, the value is blank.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
VERSION	VARCHAR(122) NOT NULL WITH DEFAULT	The version identifier for the routine. The column is a zero-length string if the value of ORIGIN is not 'I' or if the rows were created prior to Version 9.	G
OWNERTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of owner: **blank** Authorization ID **L** Role	G

SYSIBM.SYSPENDINGDDL table

The SYSIBM.SYSPENDINGDDL table contains information about which objects have pending definition changes. The entries only exist during the window between when the pending option is executed and when the utility applies these pending changes to the object.

Column name	Data type	Description	Use
DBNAME	VARCHAR(24) NOT NULL	Name of the database for the pending option.	G
TSNAME	VARCHAR(24) NOT NULL	Name of the table space for the pending option.	G
DBID	SMALLINT NOT NULL	Internal identifier of the database.	S
PSID	SMALLINT NOT NULL	Internal identifier of the table space page set descriptor.	S
OBJSCHEMA	VARCHAR(128) NOT NULL	The qualifier of the object that contains the pending option.	G
OBJNAME	VARCHAR(128) NOT NULL	Name of the object that contains the pending option.	G
OBJOBID	SMALLINT NOT NULL	Internal identifier of the object.	S
OBJTYPE	CHAR(1) NOT NULL	Type of object that is identified by OBJSCHEMA and OBJNAME. **I** The object is an index **S** The object is a table space **T** The object is a table	G
STATEMENT_TYPE	CHAR(1) NOT NULL	The type of the statement for the pending option. **A** An ALTER statement **R** A RECOVER statement	G
OPTION_ENVID	INTEGER NOT NULL	Internal identifier of the environment for the pending option.	G
OPTION_KEYWORD	VARCHAR(128) NOT NULL	If the row is inserted into this table during execution of a data definition statement, this value is the name of the pending option. If the row is inserted into this table during recovery to a prior point in time, this value is the name of the RECOVER option.	G
OPTION_VALUE	VARCHAR(4000) NOT NULL	If the row is inserted into this table during execution of a data definition statement, this value is the value of the pending option. If the row is inserted into this table during recovery to a prior point in time, this value is the value of the RECOVER option.	G
OPTION_SEQNO	SMALLINT NOT NULL	The sequence of the pending option within the statement.	G
CREATEDTS	TIMESTAMP(12) NOT NULL	Timestamp when the pending option was created.	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. See Release dependency indicators for the values.	G

Column name	Data type	Description	Use
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine- readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies. RELCREATED should be used instead.	G
ROWID	ROWID	ID to support LOB columns for source text	G
STATEMENT_TEXT	CLOB(2M) NOT NULL	The source text of the original statement for the pending option.	G
COLNAME	VARCHAR(128) NOT NULL WITH DEFAULT	The name of the column with a pending definition change.	G
PARTITION	SMALLINT NOT NULL	The partition number for the partition with a pending definition change. The value is 0 if the pending definition change is for the entire table space or index space.	G
PARTITION_ KEYWORD	VARCHAR(18) NOT NULL WITH DEFAULT	This column is populated if the PARTITION column has a non-zero value. The keyword that is associated with the PARTITION clause of the ALTER TABLE statement. For example, suppose that you issued the following statement: `ALTER TABLE ALTER PARTITION` In that case, this column contains ALTER.	G
COLUMN_ KEYWORD	VARCHAR(18) NOT NULL WITH DEFAULT	This column contains the keyword that corresponds to the column that is listed in COLNAME.	G

SYSIBM.SYSPENDINGOBJECTS table

The SYSIBM.SYSPENDINGOBJECTS table contains the name of and OBID information about objects that are the pending creation. The data sets for these objects are created but the object definition have not been materialized to the catalog. The entries in this table only exist during the time between when the names of the new objects are generated and when the catalog definition of the new objects are materialized.

Column name	Data type	Description	Use
DBNAME	VARCHAR(24) NOT NULL	Name of the database.	G
TSNAME	VARCHAR(24) NOT NULL	Name of the base table space.	G
DBID	SMALLINT NOT NULL	Internal identifier of the database.	S
PSID	SMALLINT NOT NULL	Internal identifier of the base table space page set descriptor.	S
PARTITION	SMALLINT NOT NULL	Partition number with which the object is associated.	G
COLNAME	VARCHAR(128) NOT NULL	Name of the column contained in the base table space with which the object is associated.	G
OBJSCHEMA	VARCHAR(128) NOT NULL	The qualifier of the object.	G
OBJNAME	VARCHAR(128) NOT NULL	Name of the object.	G
OBJTYPE	CHAR(1) NOT NULL	Type of object identified by OBJSCHEMA and OBJNAME. **I** The object is an index **S** The object is a table space **T** The object is a table	G
INDEXSPACE	VARCHAR(24) NOT NULL	Name of the index space. An empty string if the object is not an index.	G
OBJOBD	SMALLINT NOT NULL	Internal identifier of the object.	S
OBJPSID	SMALLINT NOT NULL	Internal identifier of the object page set descriptor, or 0 if the object does not have a page set descriptor.	S

SYSIBM.SYSPKSYSTEM table

The SYSIBM.SYSPKSYSTEM table contains zero or more rows for every package. Each row for a given package represents one or more connections to an environment in which the package could be executed.

Column name	Data type	Description	Use
LOCATION	VARCHAR(128) NOT NULL	Always contains blanks	S
COLLID	VARCHAR(128) NOT NULL	Name of the package collection.	G
NAME	VARCHAR(128) NOT NULL	Name of the package.	G
CONTOKEN	CHAR(8) NOT NULL FOR BIT DATA	Consistency token for the package. This is either: • The "level" as specified by the LEVEL option when the package's program was precompiled • The timestamp indicating when the package's program was precompiled, in an internal format.	S
SYSTEM	VARCHAR(24) NOT NULL	Environment. Values can be: **BATCH** TSO batch **CICS** Customer Information Control System **DB2CALL** DB2 call attachment facility **DLIBATCH** DLI batch support facility **IMSBMP** IMS BMP region **IMSMPP** IMS MPP and IFP region **REMOTE** remote server	G
ENABLE	CHAR(1) NOT NULL	Indicates whether the connections represented by the row are enabled or disabled: **N** Disabled **Y** Enabled	G
CNAME	VARCHAR(60) NOT NULL	Identifies the connection or connections to which the row applies. Interpretation depends on the environment specified by SYSTEM. Values can be: • Blank if SYSTEM=BATCH or SYSTEM=DB2CALL • The LU name for a database server if SYSTEM=REMOTE • Either the requester's location (if the product is DB2) or the requester's LU name enclosed in angle brackets if SYSTEM=REMOTE. • The name of a single connection if SYSTEM has any other value. CNAME can also be blank when SYSTEM is not equal to BATCH or DB2CALL. When this is so, the row applies to all servers or connections for the indicated environment.	G

Column name	Data type	Description	Use
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

SYSIBM.SYSPLAN table

The SYSIBM.SYSPLAN table contains one row for each application plan.

Column name	Data type	Description	Use
NAME	VARCHAR(24) NOT NULL	Name of the application plan.	G
CREATOR	VARCHAR(128) NOT NULL	Authorization ID of the owner of the application plan.	G
	CHAR(6) NOT NULL	Not used	N
VALIDATE	CHAR(1) NOT NULL	Whether validity checking can be deferred until run time: **B** All checking must be performed during BIND. **R** Validation is done at run time for tables, views, and privileges that do not exist at bind time.	G
ISOLATION	CHAR(1) NOT NULL	Isolation level for the plan: **R** RR (repeatable read) **T** RS (read stability) **S** CS (cursor stability) **U** UR (uncommitted read)	G
VALID	CHAR(1) NOT NULL	Whether the application plan is valid: **A** An ALTER TABLE statement changed the description of the table or base table of a view that is referred to by the application plan. For a CREATE INDEX statement involving data sharing, VALID is also marked as "A". **H** An ALTER TABLE statement changed the description of the table or base table of a view that is referred to by the application plan. **N** No **Y** Yes	G
OPERATIVE	CHAR(1) NOT NULL	Whether the application plan can be allocated: **N** No; an explicit BIND or REBIND is required before the plan can be allocated **Y** Yes	G
	CHAR(8) NOT NULL	Not used	N
PLSIZE	INTEGER NOT NULL	Size of the base section [45] of the plan, in bytes.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies. RELBOUND should be used instead.	G

45. Plans are divided into *sections*. The base section of the plan must be in the EDM pool during the entire time the application program is executing. Other sections of the plan, corresponding roughly to sets of related SQL statements, are brought into the pool as needed.

Column name	Data type	Description	Use
AVGSIZE	INTEGER NOT NULL	Average size, in bytes, of those sections[45] of the plan that contain SQL statements processed at bind time.	G
ACQUIRE	CHAR(1) NOT NULL	When resources are acquired: **A** At allocation **U** At first use	G
RELEASE	CHAR(1) NOT NULL	When resources are released: **C** At commit **D** At deallocation	G
	CHAR(1) NOT NULL	Not used	N
	CHAR(1) NOT NULL	Not used	N
	CHAR(1) NOT NULL	Not used	N
EXPLAN	CHAR(1) NOT NULL	EXPLAIN option specified for the plan; that is, whether information on the plan's statements was added to the owner's PLAN_TABLE table: **N** No **Y** Yes	G
EXPREDICATE	CHAR(1) NOT NULL	Indicates the CURRENTDATA option when the plan was bound or rebound: **B** Data currency is not required for ambiguous cursors. Allow blocking for ambiguous cursors. **C** Data currency is required for ambiguous cursors. Inhibit blocking for ambiguous cursors. **N** Blocking is inhibited for ambiguous cursors, but the plan was created before the CURRENTDATA option was available.	G
BOUNDBY	VARCHAR(128) NOT NULL WITH DEFAULT	Primary authorization ID of the binder of the plan.	G
QUALIFIER	VARCHAR(128) NOT NULL WITH DEFAULT	Implicit qualifier for the unqualified table, view, index, and alias names in the static SQL statements of the plan.	G
CACHESIZE	SMALLINT NOT NULL WITH DEFAULT	Size, in bytes, of the cache to be acquired for the plan. A value of zero indicates that no cache is used.	G
PLENTRIES	SMALLINT NOT NULL WITH DEFAULT	Number of package list entries for the plan. The negative of that number if there are rows for the plan in SYSIBM.SYSPACKLIST but the plan was bound in a prior release after fall back.	G
DEFERPREP	CHAR(1) NOT NULL WITH DEFAULT	Whether the package was last bound with the DEFER(PREPARE) option: **N** No **Y** Yes	G

Column name	Data type	Description	Use
CURRENTSERVER	VARCHAR(128) NOT NULL WITH DEFAULT	Location name specified with the CURRENTSERVER option when the plan was last bound. Blank if none was specified, implying that the first server is the local DB2 subsystem.	G
SYSENTRIES	SMALLINT NOT NULL WITH DEFAULT	Number of rows associated with the plan in SYSIBM.SYSPLSYSTEM. The negative of that number if such rows exist but the plan was bound in a prior release after fall back. A negative value or zero means that all connections are enabled.	G
DEGREE	CHAR(3) NOT NULL WITH DEFAULT	The DEGREE option used when the plan was last bound: **ANY** DEGREE(ANY) **1 or blank** DEGREE(1). Blank if the plan was migrated.	G
SQLRULES	CHAR(1) NOT NULL WITH DEFAULT	The SQLRULES option used when the plan was last bound: **D or blank** SQLRULES(DB2) **S** SQLRULES(STD) **blank** A migrated plan	G
DISCONNECT	CHAR(1) NOT NULL WITH DEFAULT	The DISCONNECT option used when the plan was last bound: **E or blank** DISCONNECT(EXPLICIT) **A** DISCONNECT(AUTOMATIC) **C** DISCONNECT(CONDITIONAL) **blank** A migrated plan	G
GROUP_MEMBER	VARCHAR(24) NOT NULL WITH DEFAULT	The DB2 data sharing member name of the DB2 subsystem that performed the most recent bind. This column is blank if the DB2 subsystem was not in a DB2 data sharing environment when the bind was performed.	G
DYNAMICRULES	CHAR(1) NOT NULL WITH DEFAULT	The DYNAMICRULES option used when the plan was last bound: **B** BIND. Dynamic SQL statements are executed with DYNAMICRULES bind behavior. **blank** RUN. Dynamic SQL statements in the plan are executed with DYNAMICRULES run behavior.	G
BOUNDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the plan was bound.	G

Column name	Data type	Description	Use
REOPTVAR	CHAR(1) NOT NULL WITH DEFAULT 'N'	Whether the access path is determined again at execution time using input variable values: **A** Bind option REOPT(AUTO) indicates that the access path is determined multiple times at execution time depending on the parameter value. **N** Bind option REOPT(NONE) indicates that the access path is determined at bind time. **Y** Bind option REOPT(ALWAYS) indicates that the access path is determined at execution time for SQL statements with variable values. **1** Bind option REOPT(ONCE) indicates that the access path is determined only once at execution time, using the first set of input variable values, regardless of how many times the same statement is executed.	G
KEEPDYNAMIC	CHAR(1) NOT NULL WITH DEFAULT 'N'	Whether prepared dynamic statements are to be purged at each commit point: **N** The bind option is KEEPDYNAMIC(NO). Prepared dynamic SQL statements are destroyed at commit or rollback. **Y** The bind option is KEEPDYNAMIC(YES). Prepared dynamic SQL statements are kept past commit or rollback.	G
PATHSCHEMAS	VARCHAR(2048) NOT NULL WITH DEFAULT	SQL path specified on the BIND or REBIND command that bound the plan. The path is used to resolve unqualified data type, function, and stored procedure names used in certain contexts. If the PATH bind option was not specified, the value in the column is a zero length string; however, DB2 uses a default SQL path of: SYSIBM, SYSFUN, SYSPROC, *plan qualifier*.	G
DBPROTOCOL	CHAR(1) NOT NULL WITH DEFAULT 'P'	Whether remote access for SQL with three-part names is implemented with DRDA or DB2 private protocol access: **D** DRDA **P** DB2 private protocol	G
FUNCTIONTS	TIMESTAMP NOT NULL WITH DEFAULT	Timestamp when the function was resolved. Set by the BIND and REBIND commands, but not by AUTOBIND.	G
OPTHINT	VARCHAR(128) NOT NULL WITH DEFAULT	Value of the OPTHINT bind option. Identifies rows in the owner.PLAN_TABLE to be used as input to DB2. Contains blanks if no rows in the owner.PLAN_TABLE are to be used as input.	G
ENCODING_CCSID	INTEGER NOT NULL WITH DEFAULT	The CCSID corresponding to the encoding scheme or CCSID as specified for the bind option ENCODING. The Encoding Scheme specified on the bind command: **ccsid** The specified or derived CCSID. **0** The default CCSID as specified on panel DSNTIPF at installation time. Used when the plan was bound prior to Version 7	G

Column name	Data type	Description	Use
IMMEDWRITE	CHAR(1) NOT NULL WITH DEFAULT	Indicates when writes of updated group buffer pool dependent pages are to be done. This option is only applicable for data sharing environments. **N** Bind option IMMEDWRITE(NO) indicates normal write activity is done. **Y** Bind option IMMEDWRITE(YES) indicates that immediate writes are done for updated group buffer pool dependent pages. **1** Bind option IMMEDWRITE(PH1) indicates that updated group buffer pool dependent pages are written at or before phase 1 commit. **blank** A migrated package.	G
RELBOUND	CHAR(1) NOT NULL WITH DEFAULT	The release when the package was bound or rebound. **blank** Bound prior to Version 7 **K** Bound on Version 7 **L** Bound on Version 8	G
	CHAR(1)	Not used.	N
REMARKS	VARCHAR(762) NOT NULL WITH DEFAULT	A character string provided by the user with the COMMENT statement.	G
CREATORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of creator: **blank** Authorization ID **L** Role	G
ROUNDING	CHAR(1) NOT NULL WITH DEFAULT	The ROUNDING option used when the plan was last bound: **C** ROUND_CEILING **D** ROUND_DOWN **F** ROUND_FLOOR **G** ROUND_HALF_DOWN **E** ROUND_HALF_EVEN **H** ROUND_HALF_UP **U** ROUND_UP **blank** The plan was created in a DB2 release prior to Version 9.	G
	DATE NOT NULL WITH DEFAULT		N
CONCUR_ACC_RES	CHAR(1) NOT NULL	Indicates the CONCURRENTACCESSRESOLUTION option when the package was bound or rebound: **blank** Not specified **U** USECURRENTLYCOMMITTED **W** WAITFOROUTCOME	G
PROGAUTH	CHAR(1) NOT NULL WITH DEFAULT 'D'	Indicates whether DB2 checks if a program is authorized to run a plan: **D** DISABLE **E** ENABLE	G

SYSIBM.SYSPLANAUTH table

The SYSIBM.SYSPLANAUTH table records the privileges that are held by users over application plans.

Column name	Data type	Description	Use
GRANTOR	VARCHAR(128) NOT NULL	Authorization ID of the user who granted the privileges.	G
GRANTEE	VARCHAR(128) NOT NULL	Authorization ID of the user who holds the privileges. Could also be PUBLIC for a grant to PUBLIC.	G
NAME	VARCHAR(24) NOT NULL	Name of the application plan on which the privileges are held.	G
	CHAR(12) NOT NULL	Internal use only	I
	CHAR(6) NOT NULL	Not used	N
	CHAR(8) NOT NULL	Not used	N
	CHAR(1) NOT NULL	Not used	N
AUTHHOWGOT	CHAR(1) NOT NULL	Authorization level of the user from whom the privileges were received. This authorization level is not necessarily the highest authorization level of the grantor. **blank** Not applicable **C** DBCTRL **D** DBADM **E** SECADM **G** ACCESSCTRL **L** SYSCTRL **M** DBMAINT **S** SYSADM	G
BINDAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can use the BIND, REBIND, or FREE subcommands against the plan: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G
EXECUTEAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can run application programs that use the application plan: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

Column name	Data type	Description	Use
GRANTEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the GRANT statement was executed.	G
GRANTEETYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of grantee: **blank** Authorization ID **L** Role	G
GRANTORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of grantor: **blank** Authorization ID **L** Role	G

SYSIBM.SYSPLANDEP table

The SYSIBM.SYSPLANDEP table records the dependencies of plans on tables, views, aliases, synonyms, table spaces, indexes, functions, and stored procedures.

Column name	Data type	Description	Use
BNAME	VARCHAR(128) NOT NULL	The name of an object the plan depends on.	G
BCREATOR	VARCHAR(128) NOT NULL	If BNAME is a table space, its database. Otherwise, the schema of BNAME. If BNAME is a role, the value is blank.	G
BTYPE	CHAR(1) NOT NULL	Type of object identified by BNAME: **A** Alias **E** INSTEAD OF trigger **F** User-defined function or cast function **G** Global temporary table **I** Index **M** Materialized query table **O** Stored procedure **P** Partitioned table space if it is defined as LARGE or with the DSSIZE parm **Q** Sequence object **R** Table space **S** Synonym **T** Table **V** View	G
DNAME	VARCHAR(24) NOT NULL	Name of the plan.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

SYSIBM.SYSPLSYSTEM table

The SYSIBM.SYSPLSYSTEM table contains zero or more rows for every plan. Each row for a given plan represents one or more connections to an environment in which the plan could be used.

Column name	Data type	Description	Use
NAME	VARCHAR(24) NOT NULL	Name of the plan.	G
SYSTEM	VARCHAR(24) NOT NULL	Environment. Values can be: **BATCH** TSO batch **DB2CALL** DB2 call attachment facility **CICS** Customer Information Control System **DLIBATCH** DLI batch support facility **IMSBMP** IMS BMP region **IMSMPP** IMS MPP or IFP region	G
ENABLE	CHAR(1) NOT NULL	Indicates whether the connections represented by the row are enabled or disabled: **N** Disabled **Y** Enabled	G
CNAME	VARCHAR(60) NOT NULL	Identifies the connection or connections to which the row applies. Interpretation depends on the environment specified by SYSTEM. Values can be: • Blank if SYSTEM=BATCH or SYSTEM=DB2CALL • The name of a single connection if SYSTEM has any other value CNAME can also be blank when SYSTEM is not equal to BATCH or DB2CALL. When this is so, the row applies to all connections for the indicated environment.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

SYSIBM.SYSQUERY table

Each SYSIBM.SYSQUERY table row identifies a SQL statement. The information is used to influence access path selection when matching statements are optimized.

Column name	Data type	Description	Use
QUERYID	BIGINT NOT NULL GENERATED BY DEFAULT AS IDENTITY	Unique identifier for the query.	G
QUERY_HASH	CHAR(16) NOT NULL FOR BIT DATA	The hash key generated by statement text.	G
SCHEMA	VARCHAR(128) NOT NULL	The default schema name for unqualified objects in the query or blank. If the query contains unqualified objects and access path hints exist for the query, the access path hints are applied only if the default schema matches the schema in the access path hint.	G
QUERY_SEC_HASH	CHAR(16) NOT NULL FOR BIT DATA	The hash key generated by the modified statement text.	G
QUERY_HASH_ VERSION	INTEGER NOT NULL	The version of the query hash.	G
SOURCE	SMALLINT NOT NULL	The source of the row: **0** Statement-level optimization hints.	G
USERFILTER	CHAR(8) NOT NULL	Filter name that is used to group a set of queries or blank.	G
	CHAR(128) NOT NULL	Internal use only.	I
PLAN_VALID	CHAR(1) NOT NULL	Whether plan hints are valid: **blank** No access path i specified for the statement, but optimization parameters exist in SYSQUERYOPTS **Y** An access path is specified in SYSQUERYPLAN for the statement. The access path is also valid if the statement has already been executed and the access path was used. **N** A an access path is specified in SYSQUERYPLAN, but the access path is invalid and not used.	G
INVALID_REASON	INTEGER NOT NULL	When PLAN_VALID is N, this column contains the reason that the access path is invalid. If PLAN_VALID is Y or blank, this column contains -1. For descriptions of the reason code values, see: +395 (DB2 Codes).	S
	VARCHAR(128) NOT NULL	Not used	N
COLLECTION	VARCHAR(128) NOT NULL	Name of the collection of the originating query or blank.	G

Column name	Data type	Description	Use
PACKAGE	VARCHAR(128) NOT NULL	Name of the package of the originating query or blank.	G
VERSION	VARCHAR(128) NOT NULL	Version of the package or blank.	G
AUTHID	VARCHAR(128) NOT NULL	Authorization ID this was in effect when the query was captured or blank.	G
BINDTIME	TIMESTAMP NOT NULL	Timestamp when the package was bound or when BIND QUERY was run	G
RELBOUND	VARCHAR(128) NOT NULL	The release of DB2 in which the package was bound, or blank. See Release dependency indicators for values.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies. RELBOUND should be used instead.	G
STMTNO	INTEGER NOT NULL	The statement number in the package. -1 when not applicable.	G
SECTNO	INTEGER NOT NULL	The section number in the package. -1 when not applicable.	G
STMTTEXT	CLOB(2M) INLINE LENGTH 2048	The text of the matching SQL statement. The value is populated from the value of the QUERY_TEXT column of the DSN_USERQUERY_TABLE table, with the following items removed: • Blanks including leading and trailing blanks, and embedded blanks that are not within literal strings between pairs of quotation mark symbols • White space, including leading and trailing white space, and white space that is not within a literal string between a pair of quotation mark symbols • SQL comments • EXPLAIN clauses	G
QUERYNO	INTEGER NOT NULL WITH DEFAULT '-1'	The query number.	G
CLIENT_USERID	VARCHAR(255)	User ID of the client.	G
CLIENT_ WRKSTNNAME	VARCHAR(255)	Name of the client workstation.	G
CLIENT_APPLNAME	VARCHAR(255)	Name of the client application.	G
SELECTVTY_ OVERRIDE	CHAR(1) NOT NULL	Whether selectivity overrides are in effect for the query: **'Y'** Selectivtiy overrides are in effect **'N'** Selectivity overrides are not in effect.	G

Column name	Data type	Description	Use
ACCESSPATH_ HINT	CHAR(1) NOT NULL	Whether access paths are specified for the matching statements: **'Y'** An access paths is specified and in effect **'N'** An access path hints is specified and in effect **blank** An access path might be specified. When the value is blank you must query the SYSIBM.SYSQUERYPLAN catalog table to determine whether an access path is specified	G
OPTION_OVERRIDE	CHAR(1) NOT NULL	Whether optimization parameters are in effect for matching statements: **'Y'** Optimization parameters are not in effect. **'N'** Optimization parameters are not in effect. **blank** Optimization parameters might be in effect. When the value is blank you must query the SYSIBM.SYSQUERYOPTS catalog table to determine whether option overrides are in effect.	G
SELECTIVITY_VALID	CHAR(1) NOT NULL	Whether selectivity overrides are valid: **blank** No selectivity overrides exist for the statement. **'Y'** Selectivity overrides exist for the query. The overrides are valid if the statement has already been executed and the overrides were used. **'N'** Selectivity overrides exist but the overrides are invalid and not used.	G

Related tasks:

Influencing access path selection (DB2 Performance)

Related reference:

Tables for influencing access path selection (DB2 Performance)

SYSIBM.SYSQUERY_AUX table

The SYSIBM.SYSQUERY_AUX table is an auxiliary table for the STMTTEXT column of the SYSIBM.SYSQUERY table.

Column name	Data type	Description	Use
STMTTEXT	CLOB(2M)	The full text of the query.	G

SYSIBM.SYSQUERYOPTS table

The SYSIBM.SYSQUERYOPTS table contains optimization parameters for the queries that are in the SYSIBM.SYSQUERY table.

Column name	Data type	Description	Use
QUERYID	BIGINT NOT NULL ON DELETE CASCADE	Unique identifier for the query. This column corresponds to the QUERYID column in the SYSIBM.SYSQUERY table.	G
COPYID	SMALLINT NOT NULL	The version of the plan hints for the query in this row. **0** Current version of the plan hints. **1** Previous version of the plan hints used by PLAN STABILITY **2** Original version of the plan hints used by PLAN STABILITY	G
REOPT	CHAR(1) NOT NULL	The value of the REOPT bind option that is in effect for the plan: **1** REOPT(ONCE) **A** REOPT(AUTO) **N** REOPT(NONE) **Y** REOPT(ALWAYS) **blank** REOPT is not specified	G
STARJOIN	CHAR(1) NOT NULL	Whether star join is enabled: **Y** Star join is enabled **N** Star join is disabled **blank** Star join is not specified	G
MAX_PAR_DEGREE	INTEGER NOT NULL	The maximum parallel degree. This column will contain a value between 0 and 254. If the value of the column is -1, the maximum parallel degree is not specified.	G
DEF_CURR_DEGREE	CHAR(3) NOT NULL	Whether query parallelism is enabled: **ONE** Query parallelism is disabled **ANY** Query parallelism is enabled **blank** Query parallelism is disabled	G
SJTABLES	INTEGER NOT NULL	The number of tables specified in a query to qualify for star join processing. If this column contains -1, star join processing is not specified.	G
	VARCHAR(128) NOT NULL	For IBM internal use only	I
GROUP_MEMBER	VARCHAR(24) NOT NULL	The group member name to which the parameters are to be applied. This column contains blank if the group member name is not specified.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

Related tasks:

Specifying optimization parameters at the statement level (DB2 Performance)

Related reference:

Tables for influencing access path selection (DB2 Performance)

DSN_USERQUERY_TABLE (DB2 Performance)

SYSIBM.SYSQUERYPLAN table

The SYSIBM.SYSQUERYPLAN table contains the plan hint information for the queries in the SYSIBM.SYSQUERY table. It correlates to the SYSIBM.SYSQUERY table by the QUERYID column. For a query, there can be up to 3 copies of plan hints stored in the SYSIBM.SYSQUERYPLAN table, distinguished by the value of the COPYID column.

Column name	Data type	Description	Use
QUERYID	BIGINT NOT NULL ON DELETE CASCADE	Unique identifier for the query. The value of QUERYID corresponds to the value of the QUERYID column in the SYSIBM.SYSQUERY column.	G
COPYID	SMALLINT NOT NULL	The version of the plan hints for the query in this row. **0** Current version of the plan hints. **1** Previous version of the plan hints used by PLAN STABILITY **2** Original version of the plan hints used by PLAN STABILITY	G
PLAN_VALID	CHAR(1) NOT NULL	Whether the plan hints are valid: **N** The plan hints are invalid **Y** The plan hints are valid	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
QBLOCKNO	SMALLINT NOT NULL	A number that identifies each query block within a query. The value of the numbers are not in any particular order, nor are they necessarily consecutive.	G
PLANNO	SMALLINT NOT NULL	The number of the step in which the query that is indicated in QBLOCKNO was processed. This column indicates the order in which the steps were executed.	G

Column name	Data type	Description	Use
METHOD	SMALLINT NOT NULL	A number that indicates the join method that is used for the step: **0** The table in this step is the first table that is accessed, a continuation of a previous table that was accessed, or a table that is not used. **1** A nested loop join is used. For each row of the current composite table, matching rows of a new table are found and joined. **2** A merge scan join is used. The current composite table and the new table are scanned in the order of the join columns, and matching rows are joined. **3** Sorts are needed by ORDER BY, GROUP BY, SELECT DISTINCT, UNION, INTERSECT, EXCEPT, a quantified predicate, or an IN predicate. This step does not access a new table. **4** A hybrid join was used. The current composite table is scanned in the order of the join-column rows of the new table. The new table is accessed using list prefetch.	G
CREATOR	VARCHAR(128) NOT NULL	The creator of the new table that is accessed in this step, blank if METHOD is 3.	G
TNAME	VARCHAR(128) NOT NULL	The name of one of the following objects: • Materialized query table • Created or declared temporary table • Materialized view • materialized table expression The value is blank if METHOD is 3. The column can also contain the name of a table in the form DSNWFQB(*qblockno*). DSNWFQB(*qblockno*) is used to represent the intermediate result of a UNION ALL, INTERSECT ALL, EXCEPT ALL, or an outer join that is materialized. If a view is merged, the name of the view does not appear. DSN_DIM_TBLX(*qblockno*) is used to the represent the work file of a star join dimension table.	G
	SMALLINT NOT NULL	Values are for IBM use only.	I

Column name	Data type	Description	Use
ACCESSTYPE	CHAR(2) NOT NULL	The method of accessing the new table: **A** Accelerated query table access. **DI** By an intersection of multiple DOCID lists to return the final DOCID list **DU** By a union of multiple DOCID lists to return the final DOCID list **DX** By an XML index scan on the index that is named in ACCESSNAME to return a DOCID list **E** By direct row access using a row change timestamp column. **H** By hash overflow index (identified in ACCESSCREATOR and ACCESSNAME) **I** By an index (identified in ACCESSCREATOR and ACCESSNAME) **IN** By an index scan when the matching predicate contains an IN predicate and the IN-list is accessed through an in-memory table. **I1** By a one-fetch index scan **M** By a multiple index scan (followed by MX, MI, MU, or MH) **MH** By the hash overflow index named in ACCESSNAME **MI** By an intersection of multiple indexes **MU** By a union of multiple indexes **MX** By an index scan on the index named in ACCESSNAME. When the access method MX follows the access method DX, DI, or DU, the table is accessed by the DOCID index by using the DOCID list that is returned by DX, DI, or DU. **N** • By an index scan when the matching predicate contains the IN keyword • By an index scan when DB2 rewrites a query using the IN keyword • By hash access with the IN keyword • By hash access when DB2 rewrites a query using the IN keyword **NR** Range list access. **P** By a dynamic pair-wise index scan **R** By a table space scan **RW** By a work file scan of the result of a materialized user-defined table function **V** By buffers for an INSERT statement within a SELECT **blank** Not applicable to the current row	G
MATCHCOLS	SMALLINT NOT NULL	For ACCESSTYPE I, I1, N, NR, MX, or DX, the number of index keys that are used in an index scan; otherwise, 0.	G
ACCESSCREATOR	VARCHAR(128) NOT NULL	For ACCESSTYPE I, I1, N, NR, MX, or DX, the creator of the index; otherwise, blank.	G
ACCESSNAME	VARCHAR(128) NOT NULL	For ACCESSTYPE I, I1, H, MH, N, NR, MX, or DX, the name of the index; for ACCESSTYPE P, DSNPJW(*mixopseqno*) is the starting pair-wise join leg in MIXOPSEQNO; otherwise, blank.	G

Column name	Data type	Description	Use
INDEXONLY	CHAR(1) NOT NULL	Indication of whether access to an index alone is enough to perform the step, or Indication of whether data too must be accessed. **Y** Yes **N** No	G
SORTN_UNIQ	CHAR(1) NOT NULL	Indication of whether the new table is sorted to remove duplicate rows. **Y** Yes **N** No	G
SORTN_JOIN	CHAR(1) NOT NULL	Indication of whether the new table is sorted for join method 2 or 4. **Y** Yes **N** No	G
SORTN_ORDERBY	CHAR(1) NOT NULL	Indication of whether the new table is sorted for ORDER BY. **Y** Yes **N** No	G
SORTN_GROUPBY	CHAR(1) NOT NULL	Indication of whether the new table is sorted for GROUP BY. **Y** Yes **N** No	G
SORTC_UNIQ	CHAR(1) NOT NULL	Indication of whether the composite table is sorted to remove duplicate rows. **Y** Yes **N** No	G
SORTC_JOIN	CHAR(1) NOT NULL	Indication of whether the composite table is sorted for join method 1, 2 or 4. **Y** Yes **N** No	G
SORTC_ORDERBY	CHAR(1) NOT NULL	Indication of whether the composite table is sorted for an ORDER BY clause or a quantified predicate. **Y** Yes **N** No	G
SORTC_GROUPBY	CHAR(1) NOT NULL	Indication of whether the composite table is sorted for a GROUP BY clause. **Y** Yes **N** No	G

Column name	Data type	Description	Use
TSLOCKMOD	CHAR(3) NOT NULL	An indication of the mode of lock that is acquired on either the new table, or its table space or table space partitions. If the isolation can be determined at bind time, the values are: **IS** Intent share lock **IX** Intent exclusive lock **S** Share lock **U** Update lock **X** Exclusive lock **SIX** Share with intent exclusive lock **N** UR isolation; no lock If the isolation level cannot be determined at bind time, the lock mode is determined by the isolation level at run time is shown by the following values. **NS** For UR isolation, no lock; for CS, RS, or RR, an S lock. **NIS** For UR isolation, no lock; for CS, RS, or RR, an IS lock. **NSS** For UR isolation, no lock; for CS or RS, an IS lock; for RR, an S lock. **SS** For UR, CS, or RS isolation, an IS lock; for RR, an S lock. The data in this column is right justified. For example, IX appears as a blank, followed by I, followed by X. If the column contains a blank, then no lock is acquired. If the access method in the ACCESSTYPE column is DX, DI, or DU, no latches are acquired on the XML index page and no lock is acquired on the new base table data page or row, nor on the XML table and the corresponding table spaces. The value of TSLOCKMODE is a blank in this case.	G
PREFETCH	CHAR(1) NOT NULL	Indication of whether data pages are to be read in advance by prefetch: **D** Optimizer expects dynamic prefetch **S** Pure sequential prefetch **L** Prefetch through a page list **blank** Unknown or no prefetch	G
COLUMN_FN_EVAL	CHAR(1) NOT NULL	When an SQL aggregate function is evaluated: **R** While the data is being read from the table or index **S** While performing a sort to satisfy a GROUP BY clause **blank** After data retrieval and after any sorts	G
MIXOPSEQ	SMALLINT NOT NULL	The sequence number of a step in a multiple index operation. **1, 2, ...** *n* For the steps of the multiple index procedure (ACCESSTYPE is MX, MI, MU, DX, DI, or DU), or the sequence number of range list access (ACCESSTYPE is 'NR'). **0** For any other rows.	G

Column name	Data type	Description	Use
ACCESS_DEGREE	SMALLINT	The number of parallel tasks or operations that are activated by a query. This value is determined at bind time; the actual number of parallel operations that are used at execution time could be different. This column contains 0 if a host variable is used. This column contains the null value if the plan or package was bound using a plan table with fewer than 43 columns. Otherwise, it can contain null if the method that it refers to does not apply.	G
ACCESS_PGROUP_ID	SMALLINT	The identifier of the parallel group for accessing the new table. A parallel group is a set of consecutive operations, executed in parallel, that have the same number of parallel tasks. This value is determined at bind time; it could change at execution time.This column contains the null value if the plan or package was bound using a plan table with fewer than 43 columns. Otherwise, it can contain null if the method that it refers to does not apply.	G
JOIN_DEGREE	SMALLINT	The number of parallel operations or tasks that are used in joining the composite table with the new table. This value is determined at bind time and can be 0 if a host variable is used. The actual number of parallel operations or tasks used at execution time could be different. This column contains the null value if the plan or package was bound using a plan table with fewer than 43 columns. Otherwise, it can contain null if the method that it refers to does not apply.	G
JOIN_PGROUP_ID	SMALLINT	The identifier of the parallel group for joining the composite table with the new table. This value is determined at bind time; it could change at execution time. This column contains the null value if the plan or package was bound using a plan table with fewer than 43 columns. Otherwise, it can contain null if the method that it refers to does not apply.	G
SORTC_PGROUP_ID	SMALLINT	The parallel group identifier for the parallel sort of the composite table. This column contains the null value if the plan or package was bound using a plan table with fewer than 43 columns. Otherwise, it can contain null if the method that it refers to does not apply.	G
SORTN_PGROUP_ID	SMALLINT	The parallel group identifier for the parallel sort of the new table. This column contains the null value if the plan or package was bound using a plan table with fewer than 43 columns. Otherwise, it can contain null if the method that it refers to does not apply.	G
PARALLELISM_ MODE	CHAR(1)	The kind of parallelism, if any, that is used at bind time: **I** Query I/O parallelism **C** Query CP parallelism This column contains the null value if the plan or package was bound using a plan table with fewer than 43 columns. Otherwise, it can contain null if the method that it refers to does not apply.	G
MERGE_ JOIN_ COLS	SMALLINT	The number of columns that are joined during a merge scan join (Method=2). This column contains the null value if the plan or package was bound using a plan table with fewer than 43 columns. Otherwise, it can contain null if the method that it refers to does not apply.	G

Column name	Data type	Description	Use
CORRELATION_ NAME	VARCHAR(128)	The correlation name of a table or view that is specified in the statement. If no correlation name exists, then the column is null. This column contains the null value if the plan or package was bound using a plan table with fewer than 43 columns. Otherwise, it can contain null if the method that it refers to does not apply.	G
PAGE_RANGE	CHAR(1) NOT NULL WITH DEFAULT	Indication of whether the table qualifies for page range screening, so that plans scan only the partitions that are needed. **Y** Yes **blank** No	G
JOIN_TYPE	CHAR(1) NOT NULL WITH DEFAULT	The type of join: **F** FULL OUTER JOIN **L** LEFT OUTER JOIN **P** Pair-wise join **S** Star join **blank** INNER JOIN or no join RIGHT OUTER JOIN converts to a LEFT OUTER JOIN when you use it, so that JOIN_TYPE contains L.	G

Column name	Data type	Description	Use
QBLOCK_TYPE	CHAR(6) NOT NULL WITH DEFAULT	For each query block, an indication of the type of SQL operation that is performed. For the outermost query, this column identifies the statement type. Possible values include: **SELECT** SELECT **INSERT** INSERT **UPDATE** UPDATE **MERGE** MERGE **DELETE** DELETE **SELUPD** SELECT with FOR UPDATE OF **DELCUR** DELETE WHERE CURRENT OF CURSOR **UPDCUR** UPDATE WHERE CURRENT OF CURSOR **CORSUB** Correlated subselect or fullselect **TRUNCA** TRUNCATE **NCOSUB** Noncorrelated subselect or fullselect **TABLEX** Table expression **TRIGGR** WHEN clause on CREATE TRIGGER **UNION** UNION **UNIONA** UNION ALL **INTERS** INTERSECT **INTERA** INTERSECT ALL **EXCEPT** EXCEPT **EXCEPTA** EXCEPT ALL	G

Column name	Data type	Description		Use
PRIMARY_ ACCESSTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates Indication of whether direct row access is attempted first:		G
		D	DB2 tries to use direct row access with a rowid column. If DB2 cannot use direct row access with a rowid column at run time, it uses the access path that is described in the ACCESSTYPE column of PLAN_TABLE.	
		T	The base table or result file is materialized into a work file, and the work file is accessed via sparse index access. If a base table is involved, then ACCESSTYPE indicates how the base table is accessed.	
		blank	DB2 does not try to use direct row access by using a rowid column or sparse index access for a work file. The value of the ACCESSTYPE column of PLAN_TABLE provides information on the method of accessing the table.	
PARENT_QBLOCKNO	SMALLINT NOT NULL	A number that indicates the QBLOCKNO of the parent query block.		G
TABLE_TYPE	CHAR(1)	The type of new table:		G
		B	Buffers for SELECT from INSERT, SELECT from UPDATE, SELECT from MERGE, or SELECT from DELETE statement.	
		C	Common table expression	
		F	Table function	
		I	The new table is generated from an IN-LIST predicate. If the IN-LIST predicate is selected as the matching predicate, it will be accessed as an in-memory table.	
		M	Materialized query table	
		Q	Temporary intermediate result table (not materialized). For the name of a view or nested table expression, a value of Q indicates that the materialization was virtual and not actual. Materialization can be virtual when the view or nested table expression definition contains a UNION ALL that is not distributed.	
		R	Recursive common table expression	
		S	Subquery (correlated or non-correlated)	
		T	Table	
		W	Work file	
		The value of the column is null if the query uses GROUP BY, ORDER BY, or DISTINCT, which requires an implicit sort.		

Column name	Data type	Description	Use
TABLE_ENCODE	CHAR(1)	The encoding scheme of the table. The possible values are: **A** ASCII **E** EBCDIC **U** Unicode **M** The table contains multiple CCSID sets	G
TABLE_SCCSID	SMALLINT NOT NULL WITH DEFAULT	The SBCS CCSID value of the table. If column TABLE_ENCODE is M, the value is 0.	G
TABLE_MCCSID	SMALLINT NOT NULL WITH DEFAULT	The mixed CCSID value of the table. If the value of the TABLE_ENCODE column is M, the value is 0. If MIXED=NO in the DSNHDECP module, the value is -2.	G
TABLE_DCCSID	SMALLINT NOT NULL WITH DEFAULT	The DBCS CCSID value of the table. If the value of the TABLE_ENCODE column is M, the value is 0. If MIXED=NO in the DSNHDECP module, the value is -2.	G
	INTEGER NOT NULL WITH DEFAULT	The values in this column are for IBM use only.	I
CTEREF	SMALLINT NOT NULL WITH DEFAULT	If the referenced table is a common table expression, the value is the top-level query block number.	G
PARENT_PLANNO	SMALLINT NOT NULL	Corresponds to the plan number in the parent query block where a correlated subquery is invoked. Or, for non-correlated subqueries, corresponds to the plan number in the parent query block that represents the work file for the subquery.	G

Column name	Data type	Description	Use
EXPANSION_REASON	CHAR(2) NOT NULL	This column applies to only static statements that reference archive tables or temporal tables. For other statements, this column is blank. Indicates the effect of the CURRENT TEMPORAL BUSINESS_TIME special register, the CURRENT TEMPORAL SYSTEM_TIME special register, and the SYSIBMADM.GET_ARCHIVE built-in global variable. These items are controlled by the BUSTIMESENSITIVE, SYSTIMESENSITIVE, and ARCHIVESENSITIVE bind options. If one of these special registers or the global variable is set to Y and the corresponding bind option is set to YES, DB2 implicitly adds certain syntax to the statement. This column indicates whether the query contains this implicit query transformation and why. This column can have one of the following values: **A** The query contains implicit query transformation as a result of the SYSIBMADM.GET_ARCHIVE built-in global variable. **B** The query contains implicit query transformation as a result of the CURRENT TEMPORAL BUSINESS_TIME special register. **S** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register. **SB** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register and the CURRENT TEMPORAL BUSINESS_TIME special register. **blank** The query does not contain implicit query transformation. **Related information:** "References to built-in global variables" on page 223 "CURRENT TEMPORAL BUSINESS_TIME" on page 194 "CURRENT TEMPORAL SYSTEM_TIME" on page 196 BIND and REBIND options (DB2 Commands)	G

Related tasks:

Specifying access paths at the statement level (DB2 Performance)

Related reference:

Tables for influencing access path selection (DB2 Performance)

PLAN_TABLE (DB2 Performance)

SYSIBM.SYSQUERYPREDICATE table

The SYSIBM.SYSQUERYPREDICATE table contains information about predicates for queries in the SYSIBM.SYSQUERY table that have been identified for extended optimization. It correlates to the SYSIBM.SYSQUERY table by the QUERYID column.

Column name	Data type	Description	Use
QUERYID	BIGINT	Identifier of the query.	S
QUERYNO	INTEGER NOT NULL	A number that identifies the statement that is being explained. The origin of the value depends on the context of the row: **For rows produced by EXPLAIN statements** The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax. **For rows not produced by EXPLAIN statements** DB2 assigns a number that is based on the line number of the SQL statement in the source program. When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in certain rare cases, the value is not guaranteed to be unique.	S
QBLOCKNO	SMALLINT NOT NULL	A number that identifies each query block within a query. The value of the numbers are not in any particular order, nor are they necessarily consecutive.	S
APPLNAME	VARCHAR(24) NOT NULL	The name of the application plan for the row. Applies only to embedded EXPLAIN statements that are executed from a plan or to statements that are explained when binding a plan. A blank indicates that the column is not applicable.	S
PROGNAME	VARCHAR(128) NOT NULL	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable.	S
PREDNO	INTEGER NOT NULL	The predicate number, a number used to identify a predicate within a query.	S

Column name	Data type	Description	Use
TYPE	CHAR(8) NOT NULL	A string used to indicate the type or the operation of the predicate. The possible values are: • 'AND' • 'OR' • 'EQUAL' • 'RANGE' • 'BETWEEN' • 'IN' • 'LIKE' • 'NOT LIKE' • 'EXISTS • 'NOTEXIST' • 'SUBQUERY' • 'HAVING' • 'OTHERS'	S
LEFT_HAND_SIDE	VARCHAR(128) NOT NULL	If the LHS of the predicate is a table column (LHS_TABNO > 0), then this column indicates the column name. Other possible values are: • 'VALUE' • 'COLEXP' • 'NONCOLEXP' • 'CORSUB' • 'NONCORSUB' • 'SUBQUERY' • 'EXPRESSION' • Blanks	S
LEFT_HAND_PNO	INTEGER NOT NULL	If the LHS of the predicate is a table column (LHS_TABNO > 0), then this column indicates the column name. Other possible values are: • 'VALUE' • 'COLEXP' • 'NONCOLEXP' • 'CORSUB' • 'NONCORSUB' • 'SUBQUERY' • 'EXPRESSION' • Blanks	S
LHS_TABNO	SMALLINT NOT NULL	If the LHS of the predicate is a table column, then this column indicates a number which uniquely identifies the corresponding table reference within a query.	S
LHS_QBNO	SMALLINT NOT NULL	If the LHS of the predicate is a table column, then this column indicates a number which uniquely identifies the corresponding table reference within a query.	S

Column name	Data type	Description	Use
RIGHT_HAND_SIDE	VARCHAR(128) NOT NULL	If the RHS of the predicate is a table column (RHS_TABNO > 0), then this column indicates the column name. Other possible values are: • 'VALUE' • 'COLEXP' • 'NONCOLEXP' • 'CORSUB' • 'NONCORSUB' • 'SUBQUERY' • 'EXPRESSION' • Blanks	S
RIGHT_HAND_PNO	INTEGER NOT NULL	If the predicate is a compound predicate (AND/OR), then this column indicates the second child predicate. However, this column is not reliable when the predicate tree consolidation happens. Use PARENT_PNO instead to reconstruct the predicate tree.	S
RHS_TABNO	SMALLINT NOT NULL	If the RHS of the predicate is a table column, then this column indicates a number which uniquely identifies the corresponding table reference within a query.	S
RHS_QBNO	SMALLINT NOT NULL	If the RHS of the predicate is a subquery, then this column indicates a number which uniquely identifies the corresponding query block within a query.	S
FILTER_FACTOR	FLOAT NOT NULL	The estimated filter factor.	S
BOOLEAN_TERM	CHAR(1) NOT NULL	Whether this predicate can be used to determine the truth value of the whole WHERE clause.	S
SEARCHARG	CHAR(1) NOT NULL	Whether this predicate can be processed by data manager (DM). If it is not, then the relational data service (RDS) needs to be used to take care of it, which is more costly.	S
JOIN	CHAR(1) NOT NULL	Whether the predicate can be used as a simple join predicate between two tables.	S
AFTER_JOIN	CHAR(1) NOT NULL	Indicates the predicate evaluation phase: **'A'** After join **'D'** During join **blank** Not applicable	S
ADDED_PRED	CHAR(1) NOT NULL	Whether it is generated by transitive closure, which means DB2 can generate additional predicates to provide more information for access path selection, when the set of predicates that belong to a query logically imply other predicates.	S
REDUNDANT_PRED	CHAR(1) NOT NULL	Whether it is a redundant predicate, which means evaluation of other predicates in the query already determines the result that the predicate provides.	S
DIRECT_ACCESS	CHAR(1) NOT NULL	Whether the predicate is direct access, which means one can navigate directly to the row through ROWID.	S
KEYFIELD	CHAR(1) NOT NULL	Whether the predicate includes the index key column of the involved table for all applicable indexes considered by DB2.	S

Column name	Data type	Description	Use
EXPLAIN_TIME	TIMESTAMP NOT NULL	The time when the EXPLAIN information was captured: **All cached statements** When the statement entered the cache, in the form of a full-precision timestamp value. **Non-cached static statements** When the statement was bound, in the form of a full precision timestamp value. **Non-cached dynamic statements** When EXPLAIN was executed, in the form of a value equivalent to a CHAR(16) representation of the time appended by 4 zeros.	S
CATEGORY	SMALLINT NOT NULL,	IBM internal use only.	S
CATEGORY_B	SMALLINT NOT NULL	IBM internal use only.	S
TEXT	VARCHAR(2000) NOT NULL	The transformed predicate text; truncated if exceeds 2000 characters.	S
PRED_ENCODE	CHAR(1) NOT NULL WITH DEFAULT	IBM internal use only.	S
PRED_CCSID	SMALLINT NOT NULL WITH DEFAULT	IBM internal use only.	S
PRED_MCCSID	SMALLINT NOT NULL WITH DEFAULT	IBM internal use only.	S
MARKER	CHAR(1) NOT NULL WITH DEFAULT	Whether this predicate includes host variables, parameter markers, or special registers.	S
PARENT_PNO	INTEGER NOT NULL	The parent predicate number. If this predicate is a root predicate within a query block, then this column is 0.	S
NEGATION	CHAR(1) NOT NULL	Whether this predicate is negated via NOT.	S
LITERALS	VARCHAR(128) NOT NULL	This column indicates the literal value or literal values separated by colon symbols.	S
CLAUSE	CHAR(8) NOT NULL	The clause where the predicate exists: **'HAVING '** The HAVING clause **'ON '** The ON clause **'WHERE '** The WHERE clause **SELECT** The SELECT clause	S
GROUP_MEMBER	VARCHAR(24) NOT NULL	The member name of the DB2 that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.	S

Column name	Data type	Description	Use
ORIGIN	CHAR(1) NOT NULL WITH DEFAULT	Indicates the origin of the predicate. **Blank** Generated by DB2 **C** Column mask **R** Row permission **U** Specified by the user	S
UNCERTAINTY	FLOAT(4) NOT NULL WITH DEFAULT	Describes the uncertainty factor of a predicate's estimated filter factor. A bigger value indicates a higher degree of uncertainty. Value zero indicates no uncertainty or uncertainty not considered.	S
SECTNOI	INTEGER NOT NULL WITH DEFAULT	The section number of the statement. The value is taken from the same column in SYSPACKSTMT or SYSSTMT tables and can be used to join tables to reconstruct the access path for the statement. This column is applicable only for static statements.	S
COLLID	VARCHAR(128) NOT NULL WITH DEFAULT	The collection ID: **DSNDYNAMICSQLCACHE** The row originates from the dynamic statement cache **DSNEXPLAINMODEYES** The row originates from an application that specifies YES for the value of the CURRENT EXPLAIN MODE special register. **DSNEXPLAINMODEEXPLAIN** The row originates from an application that specifies EXPLAIN for the value of the CURRENT EXPLAIN MODE special register.	S
VERSION	VARCHAR(122) NOT NULL WITH DEFAULT	The version identifier for the package. Applies only to an embedded EXPLAIN statement executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable.	S

Related tasks:

Overriding predicate selectivities at the statement level (DB2 Performance)

Related reference:

Tables for influencing access path selection (DB2 Performance)

DSN_PREDICAT_TABLE (DB2 Performance)

SYSIBM.SYSQUERYSEL table

The SYSIBM.SYSQUERYSEL table contains information about the selectivity of predicates for queries in the SYSIBM.SYSQUERY table that have been identified for extended optimization. It correlates to the SYSIBM.SYSQUERY table by the QUERYID column.

Column name	Data type	Description	Use
QUERYID	BIGINT	The identifier of the query.	S
QUERYNO	INTEGER NOT NULL	A number that identifies the statement that is being explained. The origin of the value depends on the context of the row: **For rows produced by EXPLAIN statements** The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax. **For rows not produced by EXPLAIN statements** DB2 assigns a number that is based on the line number of the SQL statement in the source program. When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in certain rare cases, the value is not guaranteed to be unique.	S
QBLOCKNO	SMALLINT NOT NULL	A number that identifies each query block within a query. The value of the numbers are not in any particular order, nor are they necessarily consecutive.	S
APPLNAME	VARCHAR(24) NOT NULL	The name of the application plan for the row. Applies only to embedded EXPLAIN statements that are executed from a plan or to statements that are explained when binding a plan. A blank indicates that the column is not applicable.	S
PROGNAME	VARCHAR(128) NOT NULL	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable.	S
PREDNO	INTEGER NOT NULL	Identifies the predicate	S
INSTANCE	SMALLINT NOT NULL	The selectivity instance, which is used to group related selectivities.	S
SELECTIVITY	FLOAT NOT NULL	The selectivity of the predicate.	S
WEIGHT	FLOAT (4) NOT NULL	The weight of the selectivity instance. For example, a value of .025 means that 25% of the time when a query is executed the predicate will have this selectivity.	S
ASSUMPTION	VARCHAR(128) NOT NULL	Indicates how the selectivity was estimated, or will be used: One of the following values: **'NORMAL'** Estimated using the normal selectivity assumptions. **'OVERRIDE'** To be used as input to the Optimizer and override it's selectivity estimation.	S

Column name	Data type	Description	Use
INSERT_TIME	TIMESTAMP NOT NULL GENERATED ALWAYS AS ROW CHANGE TIMESTAMP	The time when the row was inserted.	S
EXPLAIN_TIME	TIMESTAMP	The time when the EXPLAIN information was captured: **All cached statements** When the statement entered the cache, in the form of a full-precision timestamp value. **Non-cached static statements** When the statement was bound, in the form of a full precision timestamp value. **Non-cached dynamic statements** When EXPLAIN was executed, in the form of a value equivalent to a CHAR(16) representation of the time appended by 4 zeros.	S
REMARKS	VARCHAR(762)	IBM internal use only.	S

Related tasks:

Overriding predicate selectivities at the statement level (DB2 Performance)

Related reference:

Tables for influencing access path selection (DB2 Performance)

DSN_PREDICATE_SELECTIVITY table (DB2 Performance)

SYSIBM.SYSRELS table

The SYSIBM.SYSRELS table contains one row for every referential constraint.

Column name	Data type	Description	Use
CREATOR	VARCHAR(128) NOT NULL	The schema of the dependent table of the referential constraint.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the dependent table of the referential constraint.	G
RELNAME	VARCHAR(128) NOT NULL	Constraint name.	G
REFTBNAME	VARCHAR(128) NOT NULL	Name of the parent table of the referential constraint.	G
REFTBCREATOR	VARCHAR(128) NOT NULL	The schema of the parent table.	G
COLCOUNT	SMALLINT NOT NULL	Number of columns in the foreign key.	G
DELETERULE	CHAR(1) NOT NULL	Type of delete rule for the referential constraint: **A** NO ACTION **C** CASCADE **N** SET NULL **R** RESTRICT	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies. RELCREATED should be used instead.	G
RELOBID1	SMALLINT NOT NULL WITH DEFAULT	Internal identifier of the constraint with respect to the database that contains the parent table.	S
RELOBID2	SMALLINT NOT NULL WITH DEFAULT	Internal identifier of the constraint with respect to the database that contains the dependent table.	S
TIMESTAMP	TIMESTAMP NOT NULL WITH DEFAULT	Date and time the constraint was defined. If the constraint is between catalog tables prior to DB2 Version 2 Release 3, the value is '1985-04-01-00.00.00.000000.'.	G
IXOWNER	VARCHAR(128) NOT NULL WITH DEFAULT	The schema of unique non-primary index used for the parent key. '99999999' if the enforcing index has been dropped. Blank if the enforcing index is a primary index.	G
IXNAME	VARCHAR(128) NOT NULL WITH DEFAULT	Name of unique non-primary index used for a parent key. '99999999' if the enforcing index has been dropped. Blank if the enforcing index is a primary index.	G

Column name	Data type	Description	Use
ENFORCED	CHAR(1) NOT NULL WITH DEFAULT 'Y'	Enforced by the system or not: **Y** Enforced by the system **N** Not enforced by the system (trusted)	G
CHECKEXISTING-DATA	CHAR(1) NOT NULL WITH DEFAULT	Option for checking existing data: **I** Immediately check existing data. If ENFORCED = 'Y', this column will have a value of 'I'. **N** Never check existing data. If ENFORECED = 'N', this column will have a value of 'N'.	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. See Release dependency indicators for the values.	G

SYSIBM.SYSRESAUTH table

The SYSIBM.SYSRESAUTH table records CREATE IN and PACKADM ON privileges for collections; USAGE privileges for distinct types; and USE privileges for buffer pools, storage groups, and table spaces.

Column name	Data type	Description	Use
GRANTOR	VARCHAR(128) NOT NULL	Authorization ID of the user who granted the privilege.	G
GRANTEE	VARCHAR(128) NOT NULL	Authorization ID of the user who holds the privilege. Could also be PUBLIC for a grant to PUBLIC.	G
QUALIFIER	VARCHAR(128) NOT NULL	Qualifier of the table space (the database name) if the privilege is for a table space (OBTYPE='R'). The schema name of the user-defined data type if the privilege is for a distinct type (OBTYPE='D'). The schema name of the JAR file if the privilege is for a JAR file (OBTYPE='J'). The value is PACKADM if the privilege is for a collection (OBTYPE='C') and the authority held is PACKADM. Otherwise, the value is blank.	G
NAME	VARCHAR(128) NOT NULL	Name of the buffer pool, collection, DB2 storage group, distinct type, or table space. Could also be ALL when USE OF ALL BUFFERPOOLS is granted.	G
	CHAR(1) NOT NULL	Internal use only	I
AUTHHOWGOT	CHAR(1) NOT NULL	Authorization level of the user from whom the privileges were received. This authorization level is not necessarily the highest authorization level of the grantor. **blank** Not applicable **A** PACKADM (on collection *) **C** DBCTRL **D** DBADM **E** SECADM **G** ACCESSCTRL **L** SYSCTRL **M** DBMAINT **P** PACKADM (on a specific collection) **S** SYSADM **T** DATAACCESS	G
OBTYPE	CHAR(1) NOT NULL	Type of object: **B** Buffer pool **C** Collection **D** Distinct type **R** Table space **S** Storage group **J** JAR file (Java archive file)	G
	CHAR(12) NOT NULL	Internal use only	I
	CHAR(6) NOT NULL	Not used	N

Column name	Data type	Description	Use
	CHAR(8) NOT NULL	Not used	N
USEAUTH	CHAR(1) NOT NULL	Whether the privilege is held with the GRANT option: **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option The authority held is PACKADM when the OBTYPE is C (a collection) and QUALIFIER is PACKADM. The authority held is CREATE IN when the OBTYPE is C and QUALIFIER is blank.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
GRANTEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the GRANT statement was executed.	G
GRANTEETYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of grantee: **blank** Authorization ID **L** Role	G
GRANTORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of grantor: **blank** Authorization ID **L** Role	G

SYSIBM.SYSROLES table

The SYSIBM.SYSROLES table contains one row for each role.

Column name	Data type	Description	Use
NAME	VARCHAR(128) NOT NULL	The name of the role.	G
DEFINER	VARCHAR(128) NOT NULL	The authorization ID or role that defined this role listed in the NAME column.	G
DEFINERTYPE	CHAR(1) NOT NULL	The type of definer: **L** Role **blank** Authorization ID	G
CREATEDTS	TIMESTAMP NOT NULL	The time when the role is created.	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the role. See Release dependency indicators for the values.	G
REMARKS	VARCHAR(762) NOT NULL	A character string that is provided using the COMMENT statement.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies. RELCREATED should be used instead.	G

SYSIBM.SYSROUTINEAUTH table

The SYSIBM.SYSROUTINEAUTH table records the privileges that are held by users on routines. (A routine can be a user-defined function, cast function, or stored procedure.)

Column name	Data type	Description	Use
GRANTOR	VARCHAR(128) NOT NULL	Authorization ID of the user who granted the privilege.	G
GRANTEE	VARCHAR(128) NOT NULL	Authorization ID of the user who holds the privilege or the name of a plan or package that uses the privilege. Can also be PUBLIC for a grant to PUBLIC.	G
SCHEMA	VARCHAR(128) NOT NULL	Schema of the routine	G
SPECIFICNAME	VARCHAR(128) NOT NULL	Specific name of the routine. An asterisk (*) if the privilege is held on all routines in the schema.	G
GRANTEDTS	TIMESTAMP NOT NULL	Time when the GRANT statement was executed.	G
ROUTINETYPE	CHAR(1) NOT NULL	Type of routine: **F** User-defined function or cast function **P** Stored procedure	G
GRANTEETYPE	CHAR(1) NOT NULL	Type of grantee: **blank** An authorization ID **L** Role **P** An application plan or package. The grantee is a package if COLLID is not blank. **R** Internal use only	G
AUTHHOWGOT	CHAR(1) NOT NULL	Authorization level of the user from whom the privileges were received. This authorization level is not necessarily the highest authorization level of the grantor. This field is also used to indicate that the privilege was held on all schemas by the grantor. **blank** Not applicable **1** Grantor had privilege on schema.* at time of grant **E** SECADM **G** ACCESSCTRL **L** SYSCTRL **S** SYSADM **T** DATAACCESS	G
EXECUTEAUTH	CHAR(1) NOT NULL	Whether GRANTEE can execute the routine: **Y** Privilege is held without GRANT option. **G** Privilege is held with GRANT option.	G
COLLID	VARCHAR(128) NOT NULL	If the GRANTEE is a package, its collection name. Otherwise, the value is blank.	G
CONTOKEN	CHAR(8) NOT NULL FOR BIT DATA	If the GRANTEE is a package, the consistency token of the DBRM from which the package was derived. Otherwise, the value is blank.	G

Column name	Data type	Description	Use
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
GRANTORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of grantor: **blank** Authorization ID **L** Role	G

SYSIBM.SYSROUTINES table

The SYSIBM.SYSROUTINES table contains a row for every routine. (A routine can be a user-defined function, cast function, or stored procedure.)

Column name	Data type	Description	Use
SCHEMA	VARCHAR(128) NOT NULL	Schema of the routine.	G
OWNER	VARCHAR(128) NOT NULL	Owner of the routine.	G
NAME	VARCHAR(128) NOT NULL	Name of the routine.	G
ROUTINETYPE	CHAR(1) NOT NULL	Type of routine: **F** User-defined function or cast function **P** Stored procedure	G
CREATEDBY	VARCHAR(128) NOT NULL	Primary authorization ID of the user who created the routine.	G
SPECIFICNAME	VARCHAR(128) NOT NULL	Specific name of the routine.	G
ROUTINEID	INTEGER NOT NULL	Internal identifier of the routine.	S
RETURN_TYPE	INTEGER NOT NULL	Internal identifier of the result data type of the function. The column contains a -2 if the function is a table function.	S
ORIGIN	CHAR(1) NOT NULL	Origin of the routine: **E** External routine or external SQL procedure **N** Native SQL procedure **Q** SQL function **S** System-generated function **U** Sourced on user-defined function or built-in function	G
FUNCTION_TYPE	CHAR(1) NOT NULL	Type of function: **C** Aggregate function **S** Scalar function **T** Table function **blank** For a stored procedure (ROUTINETYPE = 'P')	G
PARM_COUNT	SMALLINT NOT NULL	Number of parameters for the routine.	G

Column name	Data type	Description	Use
LANGUAGE	VARCHAR(24) NOT NULL	Implementation language of the routine: • ASSEMBLE • C • COBOL • COMPJAVA • JAVA • PLI • REXX • SQL The value is blank if ROUTINETYPE = 'F' and ORIGIN is not 'E' or not 'Q'.	G
COLLID	VARCHAR(128) NOT NULL	Name of the package collection to be used when the routine is executed. A blank value indicates the package collection is the same as the package collection of the program that invoked the routine.	G
SOURCESCHEMA	VARCHAR(128) NOT NULL	If ORIGIN is 'U' and ROUTINETYPE is 'F', the schema of the source user-defined function ('SYSIBM' for a source built-in function). Otherwise, the value is blank.	G
SOURCESPECIFIC	VARCHAR(128) NOT NULL	If ORIGIN is 'U' and ROUTINETYPE is 'F', the specific name of the source user-defined function or source built-in function name. Otherwise, the value is blank.	G
DETERMINISTIC	CHAR(1) NOT NULL	The deterministic option of an external function or a stored procedure: **N** Indeterminate (results might differ with a given set of input values). **Y** Deterministic (results are consistent). **blank** ROUTINETYPE='F' and ORIGIN is not 'E' or not 'Q' (the routine is a function, but not an external function or an SQL function).	G
EXTERNAL_ACTION	CHAR(1) NOT NULL	The external action option of an external function or SQL function: **N** Function has no side effects. **E** Function has external side effects so that the number of invocations is important. **blank** ORIGIN is not 'E' or 'Q' for the function (ROUTINETYPE='F'), or it is a stored procedure (ROUTINETYPE='P').	G
NULL_CALL	CHAR(1) NOT NULL	The CALLED ON NOT NULL INPUT option of an external function or stored procedure: **N** The routine is not called if any parameter has a NULL value. **Y** The routine is called if any parameter has a NULL value. **blank** ROUTINETYPE='F' and ORIGIN is not 'E' (the routine is a function, but not an external function).	G
CAST_FUNCTION	CHAR(1) NOT NULL	Whether the routine is a cast function: **N** The routine is not a cast function. **Y** The routine is a cast function. **blank** ORIGIN is not 'E' for the function (ROUTINETYPE='F'), or it is a stored procedure (ROUTINETYPE='P'). A cast function is generated by DB2 for a CREATE TYPE statement.	G

Column name	Data type	Description	Use
SCRATCHPAD	CHAR(1) NOT NULL	The SCRATCHPAD option of an external function: **N** This function does not have a SCRATCHPAD. **Y** This function has a SCRATCHPAD. **blank** ORIGIN is not 'E' for the function (ROUTINETYPE='F'), or it is a stored procedure (ROUTINETYPE='P').	G
SCRATCHPAD_LENGTH	INTEGER NOT NULL	Length of the scratchpad if the ORIGIN is 'E' for the function (ROUTINETYPE='F') and NO SCRATCHPAD is not specified. Otherwise, the value is 0.	G
FINAL_CALL	CHAR(1) NOT NULL	The FINAL CALL option of an external function: **N** A final call will not be made to the function. **Y** A final call will be made to the function. **blank** ORIGIN is not 'E' for the function (ROUTINETYPE='F'), or it is a stored procedure (ROUTINETYPE='P').	G
PARALLEL	CHAR(1) NOT NULL	The PARALLEL option of an external function: **A** This function can be invoked by parallel tasks. **D** This function cannot be invoked by parallel tasks. **blank** ORIGIN is not 'E' for the function (ROUTINETYPE='F'), or it is a stored procedure (ROUTINETYPE='P').	G
PARAMETER_STYLE	CHAR(1) NOT NULL	The PARAMETER STYLE option of an external function or stored procedure: **D** DB2SQL. All parameters are passed to the external function or stored procedure according to the DB2SQL standard convention. **G** GENERAL. All parameters are passed to the stored procedure according to the GENERAL standard convention. **N** GENERAL CALL WITH NULLS. All parameters are passed to the stored procedure according to the GENERAL WITH NULLS convention. **J** JAVA. All parameters are passed to the function or procedure according to the conventions for JAVA and SQLJ specifications. **blank** The column is blank if the ORIGIN is not 'E' or if LANGUAGE is SQL.	G
FENCED	CHAR(1) NOT NULL	**Y** Indicates that this routine runs separately from the DB2 address space in a WLM managed DB2 address space. All user-defined routines that are not marked with Y in this column run in the DB2 address space. **blank** ORIGIN is 'Q' or ORIGIN is 'N'.	G
SQL_DATA_ACCESS	CHAR(1) NOT NULL	The SQL statements that are allowed in an external function, SQL function, or stored procedure: **C** CONTAINS SQL - Only SQL that does not read or modify data is allowed. **M** MODIFIES SQL DATA - All SQL is allowed, including SQL that reads or modifies data. **N** NO SQL - SQL is not allowed. **R** READS SQL DATA - Only SQL that reads data is allowed. **blank** Not applicable.	G

Column name	Data type	Description	Use
DBINFO	CHAR(1) NOT NULL	The DBINFO option of an external function or stored procedure: **N** No, the DBINFO parameter will not be passed to the external function or stored procedure. **Y** Yes, the DBINFO parameter will be passed to the external function or stored procedure. **blank** ORIGIN is not 'E'.	G
STAYRESIDENT	CHAR(1) NOT NULL	The STAYRESIDENT option of the routine, which determines whether the routine is to be deleted from memory when the routine ends. **N** The load module is to be deleted from memory after the routine terminates. **Y** The load module is to remain resident in memory after the routine terminates. **blank** ORIGIN is not 'E'.	G
ASUTIME	INTEGER NOT NULL	Number of CPU service units permitted for any single invocation of this routine. If ASUTIME is zero, the number of CPU service units is unlimited. The value is 0 if ROUTINETYPE = 'F' and ORIGIN is not 'E'. If a routine consumes more CPU service units than the ASUTIME value allows, DB2 cancels the routine.	G
WLM_ENVIRONMENT	VARCHAR(96) NOT NULL	Name of the WLM environment to be used to run this routine. When ORIGIN = 'N', this is the name of the WLM ENVIRONMENT FOR DEBUG MODE that is to be used when debugging a native SQL procedure. The column is blank if ROUTINETYPE = 'F' and ORIGIN is not 'E'. If the ROUTINETYPE = 'P', the value might be blank. If this value is blank the stored procedure cannot be run.	G
WLM_ENV_FOR_ NESTED	CHAR(1) NOT NULL	For nested routine calls, indicates whether the address space of the calling stored procedure or user-defined function is used to run the nested stored procedure or user-defined function: **N** The nested stored procedure or user-defined function runs in an address space other than the specified WLM environment if the calling stored procedure or user-defined function is not running in the specified WLM environment. 'WLM ENVIRONMENT name' was specified. **Y** The nested stored procedure or user-defined function runs in the environment used by the calling stored procedure or user-defined function. 'WLM ENVIRONMENT(name,*)' was specified. **blank** WLM_ENVIRONMENT is blank. The column is blank if ROUTINETYPE = 'F' and ORIGIN is not 'E'.	G
PROGRAM_TYPE	CHAR(1) NOT NULL	Indicates whether the routine runs as a Language Environment main routine or a subroutine: **M** The routine runs as a main routine. **S** The routine runs as a subroutine. **blank** ORIGIN is not 'E'.	G

Column name	Data type	Description	Use
EXTERNAL_SECURITY	CHAR(1) NOT NULL	Specifies the authorization ID to be used if the routine accesses resources protected by an external security product: **D** DB2 - The authorization ID associated with the WLM-established stored procedure address space. **U** SESSION_USER - The authorization ID of the SQL user that invoked the routine. **C** DEFINER - The authorization ID of the owner of the routine. **blank** ORIGIN is not 'E'.	G
COMMIT_ON_RETURN	CHAR(1) NOT NULL	If ROUTINETYPE = 'P', whether the transaction is always to be committed immediately on successful return (non-negative SQLCODE) from this stored procedure: **N** The unit of work is to continue. **Y** The unit of work is to be committed immediately. **A** The unit of work of the autonomous procedure is committed immediately, but other work of the calling application is not committed. If ROUTINETYPE = 'F', the value is blank.	G
RESULT_SETS	SMALLINT NOT NULL	If ROUTINETYPE = 'P', the maximum number of ad hoc result sets that this stored procedure can return. If no ad hoc result sets exist or ROUTINETYPE = 'F', the value is zero.	G
LOBCOLUMNS	SMALLINT NOT NULL	If ORIGIN = 'E' or 'Q', the number of LOB columns found in the parameter list for this user-defined function. If no LOB columns are found in the parameter list or ORIGIN is not 'E' or not 'Q', the value is 0.	S
CREATEDTS	TIMESTAMP NOT NULL	Time when the CREATE statement was executed for this routine.	G
ALTEREDTS	TIMESTAMP NOT NULL	Time when the last ALTER statement was executed for this routine.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies. RELCREATED should be used instead.	G
PARM1	SMALLINT NOT NULL	Internal use only	I
PARM2	SMALLINT NOT NULL	Internal use only	I
PARM3	SMALLINT NOT NULL	Internal use only	I

Column name	Data type	Description	Use
PARM4	SMALLINT NOT NULL	Internal use only	I
PARM5	SMALLINT NOT NULL	Internal use only	I
PARM6	SMALLINT NOT NULL	Internal use only	I
PARM7	SMALLINT NOT NULL	Internal use only	I
PARM8	SMALLINT NOT NULL	Internal use only	I
PARM9	SMALLINT NOT NULL	Internal use only	I
PARM10	SMALLINT NOT NULL	Internal use only	I
PARM11	SMALLINT NOT NULL	Internal use only	I
PARM12	SMALLINT NOT NULL	Internal use only	I
PARM13	SMALLINT NOT NULL	Internal use only	I
PARM14	SMALLINT NOT NULL	Internal use only	I
PARM15	SMALLINT NOT NULL	Internal use only	I
PARM16	SMALLINT NOT NULL	Internal use only	I
PARM17	SMALLINT NOT NULL	Internal use only	I
PARM18	SMALLINT NOT NULL	Internal use only	I
PARM19	SMALLINT NOT NULL	Internal use only	I

Column name	Data type	Description	Use
PARM20	SMALLINT NOT NULL	Internal use only	I
PARM21	SMALLINT NOT NULL	Internal use only	I
PARM22	SMALLINT NOT NULL	Internal use only	I
PARM23	SMALLINT NOT NULL	Internal use only	I
PARM24	SMALLINT NOT NULL	Internal use only	I
PARM25	SMALLINT NOT NULL	Internal use only	I
PARM26	SMALLINT NOT NULL	Internal use only	I
PARM27	SMALLINT NOT NULL	Internal use only	I
PARM28	SMALLINT NOT NULL	Internal use only	I
PARM29	SMALLINT NOT NULL	Internal use only	I
PARM30	SMALLINT NOT NULL	Internal use only	I
IOS_PER_INVOC	FLOAT NOT NULL WITH DEFAULT -1	Estimated number of I/Os that required to execute the routine. The value is -1 if the estimated number is not known.	S
INSTS_PER_INVOC	FLOAT NOT NULL WITH DEFAULT -1	Estimated number of machine instructions that required to execute the routine. The value is -1 if the estimated number is not known.	S
INITIAL_IOS	FLOAT NOT NULL WITH DEFAULT -1	Estimated number of I/O's that are performed the first time or the last time the routine is invoked. The value is -1 if the estimated number is not known.	S
INITIAL_INSTS	FLOAT NOT NULL WITH DEFAULT -1	Estimated number of machine instructions that are performed the first time or the last time the routine is invoked. The value is -1 if the estimated number is not known.	S

Column name	Data type	Description	Use
CARDINALITY	FLOAT NOT NULL WITH DEFAULT -1	The predicted cardinality of the routine, -1 to trigger the use of the default value (10,000).	S
RESULT_COLS	SMALLINT NOT NULL DEFAULT 1	For a table function, the number of columns in the result table. Otherwise, the value is 1.	S
EXTERNAL_NAME	VARCHAR(762) NOT NULL	The path/module/function that DB2 should load to execute the routine. The column is blank if ROUTINETYPE = 'F' and ORIGIN is not 'E'.	G
	VARCHAR(150) NOT NULL FOR BIT DATA	Internal use only	I
RUNOPTS	VARCHAR(762) NOT NULL	The Language Environment run time options to be used for this routine. An empty string indicates that the installation default Language Environment run time options are to be used. The column is blank if ROUTINETYPE = 'F' and ORIGIN is not 'E'.	G
REMARKS	VARCHAR(762) NOT NULL	A character string provided by the user with the COMMENT statement.	G
JAVA_SIGNATURE	VARCHAR(3072) NOT NULL WITH DEFAULT	The signature of the JAR file. **blank** When PARAMETER STYLE is not JAVA. The column is also blank if ROUTINETYPE = 'F' and ORIGIN is not 'E'.	G
CLASS	VARCHAR(384) NOT NULL WITH DEFAULT	The class name contained in the JAR file. **blank** When PARAMETER STYLE is not JAVA. The column is also blank if ROUTINETYPE = 'F' and ORIGIN is not 'E'.	G
JARSCHEMA	VARCHAR(128) NOT NULL WITH DEFAULT	The schema of the JAR file. **blank** When PARAMETER STYLE is not JAVA. The column is also blank if ROUTINETYPE = 'F' and ORIGIN is not 'E'.	G
JAR_ID	VARCHAR(128) NOT NULL WITH DEFAULT	The name of the JAR file. **blank** When PARAMETER STYLE is not JAVA. The column is also blank if ROUTINETYPE = 'F' and ORIGIN is not 'E'.	G
SPECIAL_REGS	CHAR(1) NOT NULL WITH DEFAULT 'I'	The SPECIAL REGISTER option for a routine. **I** INHERIT SPECIAL REGISTERS **D** DEFAULT SPECIAL REGISTERS **blank** ROUTINETYPE = 'F' and ORIGIN is not 'E' or not 'Q'.	G
NUM_DEP_MQTS	SMALLINT NOT NULL WITH DEFAULT	Number of dependent materialized query tables. The value is 0 if the row does not describe a user-defined table function, or if no materialized query tables are defined on the table function.	G

Column name	Data type	Description	Use
MAX_FAILURE	SMALLINT NOT NULL WITH DEFAULT -1	Allowable failures for this routine (0-32767). If zero is specified, the routine will never be stopped. If no value is specified for this routine, the default will be -1 to indicate that the DB2 installation parameter (STORMXAB) will be used.	G
PARAMETER_CCSID	INTEGER NOT NULL WITH DEFAULT	A CCSID that specifies how character, graphic, date, time, and timestamp data types for system generated parameters to the routine such as message tokens and DBINFO should be passed. The value is dependent on the encoding scheme specified implicitly or explicitly for the PARAMETER CCSID clause defined at the system for that encoding scheme. The following list describes the CCSID for each encoding scheme: **ASCII** If mixed data is allowed, this CCSID is for mixed ASCII data, SBCS data uses the corresponding SBCS CCSID, and graphic data uses the corresponding DBCS CCSID. Otherwise, this CCSID is for SBCS ASCII data. **EBCDIC** If mixed data is allowed, this CCSID is for mixed EBCDIC data, SBCS data uses the corresponding SBCS CCSID, and graphic data uses the corresponding DBCS CCSID. Otherwise, this is the CCSID for SBCS EBCDIC data. **UNICODE** This CCSID is for mixed data (1208). A value of zero means that the CCSIDs used are those CCSIDs for the encoding scheme of other string or datetime parameters in the parameter list or RETURNS clause CCSID clauses, or the value in the DEF ENCODING SCHEME on installation panel DSNTIPF.	G
VERSION	VARCHAR(122) NOT NULL WITH DEFAULT	The version identifier for a native SQL procedure (indicated by the value 'N' in the column ORIGIN) or a non-inline SQL scalar function (indicated by the value 'Q' in the column ORIGIN and 'N' in the column INLINE). A zero length string for the rows that are created prior to Version 9 and for the rows that correspond to neither native SQL procedures or non-inline SQL scalar functions.	G
CONTOKEN	CHAR(8) NOT NULL WITH DEFAULT FOR BIT DATA	The consistency token for the routine. The column is set to X'20' if the value of ORIGIN is not 'N'	G
ACTIVE	CHAR(1) NOT NULL WITH DEFAULT	Identifies the active version of the routine: **Y** The routine is the active version. **N** The routine is not the active version. **blank** The value of ORIGIN is not 'N' or the row was created prior to Version 9.	G

Column name	Data type	Description	Use
DEBUG_MODE	CHAR(1) NOT NULL WITH DEFAULT	Identifies whether or not this routine is enabled for debugging: **1** This routine is enabled for debugging and can be debugged in a client debug session using the DB2 Unified Debugger. **0** This routine is not enabled for debugging. **N** This routine can never be enabled for debugging. **blank** The LANGUAGE is not specified as JAVA, the value of ORIGIN is not 'N', or the row was created prior to Version 9.	G
TEXT_ENVID	INTEGER NOT NULL WITH DEFAULT	Internal identifier of the environment. The column is 0 if the value of ORIGIN is not 'N' or if the row was created prior to Version 9.	G
TEXT_ROWID	ROWID NOT NULL GENERATED ALWAYS	ID to support LOB columns for source text.	G
TEXT	CLOB(2M) NOT NULL WITH DEFAULT	The source text of the CREATE statement or the ALTER statement with the body for the routine. The column is a zero-length string if the value of ORIGIN is not 'N' or if the row was created prior to Version 9.	G
OWNERTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of owner: **blank** Authorization ID **L** Role	G
PARAMETER_ VARCHARFORM	INTEGER NOT NULL WITH DEFAULT	A non-zero value that indicates the actual representation, to a LANGUAGE C routine, of any varying length string parameter that appears in the parameter list or RETURNS clause for that routine.	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. Blank if created prior to Version 9. See Release dependency indicators for all other values.	G
PACKAGEPATH	VARCHAR(4096)	The value of the PACKAGE PATH option of the CREATE FUNCTION, CREATE PROCEDURE, ALTER FUNCTION, or ALTER PROCEDURE statement that created or last changed the routine. PACKAGE PATH identifies the package path to use when the routine is executed. A blank value indicates the package path is the same as the package path of the program that invoked the routine.	G
SECURE	CHAR(1) NOT NULL WITH DEFAULT 'N'	Indicates if the routine is secured: **N** The routine is not secured **Y** The routine is secured	G

Column name	Data type	Description		Use
INLINE	CHAR(1) NOT NULL WITH DEFAULT	Specifies if the SQL function is inline:		G
		Y	The SQL function is inline when referenced. No package is associated with this type of routine.	
		N	The SQL function has an associated package.	
		blank	Not an SQL function (the ORIGIN column has a value other than 'Q')	
	BLOB(1G) NOT NULL WITH DEFAULT	Internal use only		I
SYSTEM_DEFINED	CHAR(1) NOT NULL WITH DEFAULT	Identifies whether this routine is system defined:		G
		blank	This routine is not system defined	
		S	This routine is system defined	

SYSIBM.SYSROUTINESTEXT table

The SYSIBM.SYSROUTINESTEXT is an auxiliary table for the TEXT column of SYSIBM.SYSROUTINES and is required to hold the LOB data.

Column name	Data type	Description	Use
TEXT	CLOB(2M) NOT NULL WITH DEFAULT	The source text of the CREATE PROCEDURE statement for the routine. TEXT can also hold the source text of the ALTER PROCEDURE statement for the routine if the routine is a native SQL procedure and the SQL procedure body is included in the ALTER PROCEDURE statement.	G

SYSIBM.SYSROUTINES_OPTS table

The SYSIBM.SYSROUTINES_OPTS table Contains a row for each generated routine, such as one created by DB2 for z/OS Procedure Processor DSNTPSMP, that records the build options for the routine.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
SCHEMA	VARCHAR(128) NOT NULL	Schema of the routine.	G
ROUTINENAME	VARCHAR(128) NOT NULL	Name of the routine.	G
BUILDDATE	DATE NOT NULL WITH DEFAULT	Date the routine was built.	G
BUILDTIME	TIME NOT NULL WITH DEFAULT	Time the routine was built.	G
BUILDSTATUS	CHAR(1) NOT NULL WITH DEFAULT 'C'	Whether this version of the routine's options is the current version.	G
BUILDSCHEMA	VARCHAR(128) NOT NULL	Schema name for BUILDNAME.	G
BUILDNAME	VARCHAR(128) NOT NULL	Procedure used to create the routine.	G
BUILDOWNER	VARCHAR(128) NOT NULL	Authorization ID used to create the routine.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
PRECOMPILE_OPTS	VARCHAR(765) NOT NULL WITH DEFAULT	SQL processing (precompiler or coprocessor) options used to build the routine.	G
COMPILE_OPTS	VARCHAR(765) NOT NULL WITH DEFAULT	Compiler options used to build the routine.	G
PRELINK_OPTS	VARCHAR(765) NOT NULL WITH DEFAULT	Prelink-edit options used to build the routine.	G

Column name	Data type	Description	Use
LINK_OPTS	VARCHAR(765) NOT NULL WITH DEFAULT	Link-edit options used to build the routine.	G
BIND_OPTS	VARCHAR(3072) NOT NULL WITH DEFAULT	Bind options used to build the routine.	G
SOURCEDSN	VARCHAR(765) NOT NULL WITH DEFAULT	Name of the source data set.	G
DEBUG_MODE	CHAR(1) NOT NULL	Debugging is on or off for this object. **0** Debugging is off. Default and value on migration are both 0. **1** Debugging is on.	G

SYSIBM.SYSROUTINES_TREE table

The SYSIBM.SYSROUTINES_TREE table is an auxiliary table for the PTREE column of the SYSIBM.SYSROUTINES table.

Column name	Data type	Description	Use
PTREE	BLOB(1G) NOT NULL WITH DEFAULT	Internal use only.	I

SYSIBM.SYSROUTINES_SRC table

The SYSIBM.SYSROUTINES_SRC table contains source for generated routines, such as those created by DB2 for z/OS Procedure Processor DSNTPSMP.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
SCHEMA	VARCHAR(128) NOT NULL	Schema of the routine.	G
ROUTINENAME	VARCHAR(128) NOT NULL	Name of the routine.	G
BUILDDATE	DATE NOT NULL WITH DEFAULT	Date the routine was built.	G
BUILDTIME	TIME NOT NULL WITH DEFAULT	Time the routine was built.	G
BUILDSTATUS	CHAR(1) NOT NULL WITH DEFAULT 'C'	Whether this version of the routine's source is the current version.	G
SEQNO	INTEGER NOT NULL	Number of the source statement piece in CREATESTMT.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
CREATESTMT	VARCHAR(7500) NOT NULL	Routine source statement.	G

SYSIBM.SYSSCHEMAAUTH table

The SYSIBM.SYSSCHEMAAUTH table contains one or more rows for each user that is granted a privilege on a particular schema in the database.

Column name	Data type	Description	Use
GRANTOR	VARCHAR(128) NOT NULL	Authorization ID of the user who granted the privileges or SYSADM.	G
GRANTEE	VARCHAR(128) NOT NULL	Authorization ID of the user or group who holds the privileges. Can also be PUBLIC for a grant to PUBLIC.	G
SCHEMANAME	VARCHAR(128) NOT NULL	Name of the schema or '*' for all schemas.	G
AUTHHOWGOT	CHAR(1) NOT NULL	Authorization level of the user from whom the privileges were received. This authorization level is not necessarily the highest authorization level of the grantor. This field is also used to indicate that the privilege was held on all schemas by the grantor. **1** Grantor had privilege on all schemas at time of grant **E** SECADM **G** ACCESSCTRL **L** SYSCTRL **S** SYSADM	G
CREATEINAUTH	CHAR(1) NOT NULL	Indicates whether grantee holds CREATEIN privilege on the schema: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G
ALTERINAUTH	CHAR(1) NOT NULL	Indicates whether grantee holds ALTERIN privilege on the schema: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G
DROPINAUTH	CHAR(1) NOT NULL	Indicates whether grantee holds DROPIN privilege on the schema: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G
GRANTEDTS	TIMESTAMP NOT NULL	Time when the GRANT statement was executed.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
GRANTEETYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of grantee: **blank** Authorization ID **L** Role	G

Column name	Data type	Description	Use
GRANTORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of grantor: **blank** Authorization ID **L** Role	G

SYSIBM.SYSSEQUENCEAUTH table

The SYSIBM.SYSSEQUENCEAUTH table records the privileges that are held by users over sequences.

Column name	Data type	Description	Use
GRANTOR	VARCHAR(128) NOT NULL	Authorization ID of the user who granted the privileges.	G
GRANTEE	VARCHAR(128) NOT NULL	Authorization ID of the user or group that holds the privileges or the name of an application plan or package that uses the privileges. PUBLIC for a grant to PUBLIC.	G
SCHEMA	VARCHAR(128) NOT NULL	Schema of the sequence.	G
NAME	VARCHAR(128) NOT NULL	Name of the sequence.	G
GRANTEETYPE	CHAR(1) NOT NULL	Type of grantee: **blank** An authorization ID. **L** Role **P** An application plan or package. The grantee is a package if COLLID is not blank. **R** Internal use only.	G
AUTHHOWGOT	CHAR(1) NOT NULL	Authorization level of the user from whom the privileges were received. This authorization level is not necessarily the highest authorization level of the grantor: **blank** Not applicable **E** SECADM **G** ACCESSCTRL **L** SYSCTRL **S** SYSADM **T** DATAACCESS	G
ALTERAUTH	CHAR(1) NOT NULL	Indicates whether grantee holds ALTER privilege on the sequence: **blank** Privilege is not held. **G** Privilege is held with the GRANT option. **Y** Privilege is held without the GRANT option.	G
USEAUTH	CHAR(1) NOT NULL	Indicates whether grantee holds USAGE privilege on the sequence: **blank** Privilege is not held. **G** Privilege is held with the GRANT option. **Y** Privilege is held without the GRANT option.	G
COLLID	VARCHAR(128) NOT NULL	If the GRANTEE is a package, its collection name. Otherwise, a string of length zero.	G
CONTOKEN	CHAR(8) NOT NULL FOR BIT DATA	If the GRANTEE is a package, the consistency token of the DBRM from which the package was derived. Otherwise, blank.	G
GRANTEDTS	TIMESTAMP NOT NULL	Time when the GRANT statement was executed.	G

Column name	Data type	Description	Use
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
GRANTORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of grantor: **blank** Authorization ID **L** Role	G

SYSIBM.SYSSEQUENCES table

The SYSIBM.SYSSEQUENCES table contains one row for each identity column or user-defined sequence.

Column name	Data type	Description	Use
SCHEMA	VARCHAR(128) NOT NULL	Schema of the alias or sequence. For an identity column, the value of TBCREATOR from the SYSCOLUMNS entry for the column.	G
OWNER	VARCHAR(128) NOT NULL	Owner of the alias or sequence. For an identity column, the value of TBCREATOR from the SYSCOLUMNS entry for the column.	G
NAME	VARCHAR(128) NOT NULL	Name of the alias, identity column, or sequence. The name for an identity column is generated by DB2.	G
SEQTYPE	CHAR(1) NOT NULL	Type of sequence object: **A** Alias for a sequence **I** An identity column **S** A user-defined sequence **X** An implicitly created DOCID column for a base table that contains XML data.	G
SEQUENCEID	INTEGER NOT NULL	Internal identifier of the alias, identity column, or sequence.	G
CREATEDBY	VARCHAR(128) NOT NULL	Primary authorization ID of the user who created the alias, identity column, or sequence.	G
INCREMENT	DECIMAL(31,0) NOT NULL	Increment value (positive or negative, within INTEGER scope). The value is 0 if the row describes an alias.	G
START	DECIMAL(31,0) NOT NULL	Start value. The value is 0 if the row describes an alias.	G
MAXVALUE	DECIMAL(31,0) NOT NULL	Maximum value allowed for the identity column or sequence. The value is 0 if the row describes an alias.	G
MINVALUE	DECIMAL(31,0) NOT NULL	Minimum value allowed for the identity column or sequence. The value is 0 if the row describes an alias.	G
CYCLE	CHAR(1) NOT NULL	Whether cycling will occur when a boundary is reached: **N** No **Y** Yes, cycling will occur **blank** The row describes an alias	G
CACHE	INTEGER NOT NULL	Number of sequence values to preallocate in memory for faster access. A value of 0 indicates that values are not to be preallocated. The value is 0 if the row describes an alias.	G

Column name	Data type	Description	Use
ORDER	CHAR(1) NOT NULL	Whether the values must be generated in order: **Y** Yes **N** No **R** The values must be generated in pseudo-random order for an XML document ID column that was created when subsystem parameter XML_RANDOMIZE_DOCID was set to YES. **blank** The row describes an alias	G
DATATYPEID	INTEGER NOT NULL	For a built-in data type, the internal ID of the built-in type. For a distinct type, the internal ID of the distinct type. The value is 0 if the row describes an alias.	S
SOURCETYPEID	INTEGER NOT NULL	For a built-in data type, 0. For a distinct type, the internal ID of the built-in data type upon which the distinct type is based. The value is 0 if the row describes an alias.	S
CREATEDTS	TIMESTAMP NOT NULL	Timestamp of the creation of the alias, identity column, or sequence.	G
ALTEREDTS	TIMESTAMP NOT NULL	Timestamp when the last ALTER statement was executed for this alias, identity column, or sequence.	G
MAXASSIGNEDVAL	DECIMAL(31,0)	Last possible assigned value. Initialized to null when the object is created. Updated each time the next chunk of *n* values is cached, where *n* is the value for CACHE. The value is 0 if the row describes an alias.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies. RELCREATED should be used instead.	G
REMARKS	VARCHAR(762) NOT NULL	Character string provided by user with the COMMENT statement. The value is blank for an identity column.	G
PRECISION	SMALLINT NOT NULL WITH DEFAULT	The precision defined for a sequence with a decimal or numeric type. The value is 5 for SMALLINT, 10 for INTEGER, or the actual precision specified by the user for the decimal data type. The value is 0 for rows created prior to Version 8. The value is 0 if the row describes an alias.	G
RESTARTWITH	DECIMAL(31,0) NULLABLE WITH DEFAULT	The RESTART WITH value specified for a sequence during ALTER. The value is NULL for the following: • There have been no ALTER with RESTART WITH • The row describes an alias	G

Column name	Data type	Description	Use
OWNERTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of owner: **blank** Authorization ID **L** Role	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. Blank if created prior to Version 9. See Release dependency indicators for all other values.	G
SEQSCHEMA	VARCHAR(128) NOT NULL WITH DEFAULT	The schema of the target sequence.	G
SEQNAME	VARCHAR(128) NOT NULL WITH DEFAULT	The name of the target sequence.	G

SYSIBM.SYSSEQUENCESDEP table

The SYSIBM.SYSSEQUENCESDEP table records the dependencies of identity columns and sequences.

Column name	Data type	Description	Use
BSEQUENCEID	INTEGER NOT NULL	Internal identifier of the identity column or sequence.	G
DCREATOR	VARCHAR(128) NOT NULL	The owner of the object that is dependent on this identity column or sequence.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
DNAME	VARCHAR(128) NOT NULL	Name of the object that is dependent on this identity column or sequence.	G
DCOLNAME	VARCHAR(128) NOT NULL	Name of the identity column. Blank for SQL function rows.	G
DTYPE	CHAR(1) NOT NULL WITH DEFAULT 'I'	The type of object that is dependent on this sequence: **F** SQL function **I** Identity column **X** Implicit DOCID column that is created on a base table with XML **blank** Represents an identity column created prior to Version 8	G
BSCHEMA	VARCHAR(128) NOT NULL WITH DEFAULT	The schema name of the sequence, will be a string of length zero for an object created prior to Version 8.	G
BNAME	VARCHAR(128) NOT NULL WITH DEFAULT	The sequence name (generated by DB2 for an identity column), will be a string of length zero for an object created prior to Version 8.	G
DSCHEMA	VARCHAR(128) NOT NULL WITH DEFAULT	The qualifier of the object that is dependent on this sequence, will be a string of length zero for an object created prior to Version 8.	G
DOWNER	VARCHAR(128) NOT NULL WITH DEFAULT	The owner of the object that is dependent on this sequence. This will be a string of length zero for an object that was created prior to Version 9.	G
DOWNERTYPE	CHAR(1) NOT NULL WITH DEFAULT	The type of owner: **Blank** An authorization ID **L** A role	G

SYSIBM.SYSSTATFEEDBACK table

The SYSIBM.SYSTATFEEDBACK table contains information about missing or conflicting catalog statistics for SQL statements.

The following values control the collection of statistics feedback data in the SYSIBM.SYSSTATFEEDBACK catalog table:

- The STATFDBK_SCOPE subsystem parameter controls whether the data is collected, and whether it is collected only for static SQL statements, only for dynamic SQL statements, or for both.
- The STATSINT subsystem parameter controls when and how frequently the data is externalized.
- The STATS_FEEDBACK column of the SYSIBM.SYSTABLES catalog table controls whether the data is collected for a particular table.

The RUSNTATS utility removes data from the SYSIBM.SYSSTATFEEDBACK catalog table when the recommended statistics are collected.

Column name	Data type	Description	Use
TBCREATOR	VARCHAR(128)	The creator of the table.	S
TBNAME	VARCHAR(128)	The name of the table.	S
IXCREATOR	VARCHAR(128)	The creator of the index.	S
IXNAME	VARCHAR(128)	The name of the index.	S
COLNAME	VARCHAR(128)	The name of the column.	S
NUMCOLUMNS	SMALLINT	The number of columns in the column group.	S
COLGROUPCOLNO	VARCHAR(254) FOR BIT DATA	A hex representation that identifies the set of columns associated with the statistics. If the statistics are only associated with a single column, the field contains a zero length. Otherwise, the field is an array of SMALLINT column numbers with a dimension equal to the value in NUMCOLUMNS.	S
TYPE	CHAR(1)	The type of statistic to collect: **'C'** Cardinality. **'F'** Frequency. **'H'** Histogram. **'I'** Index. **'T'** Table.	I
DBNAME	VARCHAR(24)	The name of the database.	S
TSNAME	VARCHAR(24)	The name of the table space.	S

Column name	Data type	Description	Use
REASON	CHAR(8)	The reason that the statistic was recommend: **'BASIC'** A basic statistical value for a column table or index is missing. **'KEYCARD'** The cardinalities of index key columns are missing. **'LOWCARD'** The cardinality of the column is a low value, which indicates that data skew is likely. **'NULLABLE'** Distribution statistics are not available for a nullable column. **'DEFAULT'** A predicate references a value that is probably a default value. **'RANGEPRD'** Histogram statistics are not available for a range predicate. **'PARALLEL'** Parallelism could be improved by uniform partitioning of key ranges. **'CONFLICT'** Another statistic conflicts with this statistic. **'COMPFFIX'** Multi-column cardinality statistics are needed for an index compound filter factor.	S
BLOCK_RUNSTATS	CHAR(1)	Whether the row is used when optimization tools collect statistics based on the recommendations. DB2 inserts a blank value in this column for all new rows. DB2 does not refer to or change the value of this column. This is an updatable column.	S
REMARKS	VARCHAR(254)	Free form text for extensibility.	S
LASTDATE	DATE	The last date that this statistics recommendation was updated by DB2.	S

Related tasks:

- Maintaining statistics in the catalog (DB2 Performance)

Related reference:

- DSN_STAT_FEEDBACK (DB2 Performance)
- Statistics used for access path selection (DB2 Performance)
- STATISTICS FEEDBACK field (STATFDBK_SCOPE subsystem parameter) (DB2 Installation and Migration)
- REAL TIME STATS field (STATSINT subsystem parameter) (DB2 Installation and Migration)
- "SYSIBM.SYSTABLES table" on page 2396
- RUNSTATS (DB2 Utilities)

SYSIBM.SYSSTMT table

The SYSIBM.SYSSTMT table contains one or more rows for each SQL statement of each DBRM.

Column name	Data type	Description	Use
NAME	VARCHAR(24) NOT NULL	Name of the DBRM.	G
PLNAME	VARCHAR(24) NOT NULL	Name of the application plan.	G
PLCREATOR	VARCHAR(128) NOT NULL	Authorization ID of the owner of the application plan.	G
SEQNO	INTEGER NOT NULL	Sequence number of this row with respect to a statement of the plan. Rows in which the values of SEQNO, STMTNO, and SECTNO are zero are for internal use. The numbering starts with zero.	G
STMTNO	SMALLINT NOT NULL	The statement number of the statement in the source program. A statement number greater than 32767 is stored as zero. If the value is zero, see STMTNOI for the statement number. Rows in which the values of SEQNO, STMTNO, and SECTNO are zero are for internal use.	G
SECTNO	SMALLINT NOT NULL	The section number of the statement. Rows in which the values of SEQNO, STMTNO, and SECTNO are zero are for internal use.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
TEXT	VARCHAR(3800) NOT NULL FOR BIT DATA	Text or portion of the text of the SQL statement.	S
ISOLATION	CHAR(1) NOT NULL WITH DEFAULT	Isolation level for the SQL statement: **R** RR (repeatable read) **T** RS (read stability) **S** CS (cursor stability) **U** UR (uncommitted read) **L** RS isolation, with a *lock-clause* **X** RR isolation, with a *lock-clause* **blank** The WITH clause was not specified on this statement. The isolation level is recorded in SYSPACKAGE.ISOLATION and in SYSPLAN.ISOLATION.	G

Column name	Data type	Description	Use
STATUS	CHAR(1) NOT NULL WITH DEFAULT	Status of binding the statement: **A** Distributed - statement uses DB2 private protocol access. The statement will be parsed and executed at the server using defaults for input variables during access path selection. **B** Distributed - statement uses DB2 private protocol access. The statement will be parsed and executed at the server using values for input variables during access path selection. **C** Compiled - statement was bound successfully using defaults for input variables during access path selection. **D** Distributed - statement references a remote object using a three-part name. DB2 will implicitly use DRDA access either because the DBPROTOCOL bind option was not specified (defaults to DRDA), or the bind option DBPROTOCOL(DRDA) was explicitly specified. This option allows the use of three-part names with DRDA access but it requires that the package be bound at the target remote site. **E** Explain - statement is an SQL EXPLAIN statement. The explain is done at bind time using defaults for input variables during access path selection. **F** Parsed - statement did not bind successfully and VALIDATE(RUN) was used. The statement will be rebound at execution time using values for input variables during access path selection. **G** Compiled - statement bound successfully, but REOPT is specified. The statement will be rebound at execution time using values for input variables during access path selection. **H** Parsed - statement is either a data definition statement or a statement that did not bind successfully and VALIDATE(RUN) was used. The statement will be rebound at execution time using defaults for input variables during access path selection. Data manipulation statements use defaults for input variables during access path selection. **I** Indefinite - statement is dynamic. The statement will be bound at execution time using defaults for input variables during access path selection.	S

Column name	Data type	Description	Use
STATUS		**J** Indefinite - statement is dynamic. The statement will be bound at execution time using values for input variables during access path selection. **K** Control - CALL statement. **L** Bad - the statement has some allowable error. The bind continues but the statement cannot be executed. **M** Parsed - statement references a table that is qualified with SESSION and was not bound because the table reference could be for a declared temporary table that will not be defined until the package or plan is run. The SQL statement will be rebound at execution time using values for input variables during access path selection. **blank** The statement is non-executable, or was bound in a DB2 release prior to Version 5.	
ACCESSPATH	CHAR(1) NOT NULL WITH DEFAULT	For static statements, indicates if the access path for the statement is based on user-specified optimization hints. A value of 'H' indicates that optimization hints were used. A blank value indicates that the access path was determined without the use of optimization hints, or that there is no access path associated with the statement. For dynamic statements, the value is blank.	G
STMTNOI	INTEGER NOT NULL WITH DEFAULT	If the value of STMTNOI is not zero, the column contains the statement number of the statement in the source program.	G
SECTNOI	INTEGER NOT NULL WITH DEFAULT	The section number of the statement.	G
EXPLAINABLE	CHAR(1) NOT NULL WITH DEFAULT	Contains one of the following values: **Y** Indicates that the SQL statement can be used with the EXPLAIN function and might have rows describing its access path in the owner.PLAN_TABLE. **N** Indicates that the SQL statement does not have any rows describing its access path in the owner.PLAN_TABLE. **blank** Indicates that the SQL statement was bound prior to Version 7.	G
QUERYNO	INTEGER NOT NULL WITH DEFAULT –1	The query number of the SQL statement in the source program. SQL statements bound prior to Version 7 have a default value of –1. Statements bound in Version 7 or later use the value specified on the QUERYNO clause on SELECT, UPDATE, INSERT, DELETE, EXPLAIN, and DECLARE CURSOR statements. If the QUERYNO clause is not specified, the query number is set to the statement number.	G

Column name	Data type	Description	Use
PLCREATORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of creator: **blank** Authorization ID **L** Role	G

SYSIBM.SYSSTOGROUP table

The SYSIBM.SYSSTOGROUP table contains one row for each storage group.

Column name	Data type	Description	Use
NAME	VARCHAR(128) NOT NULL	Name of the storage group.	G
CREATOR	VARCHAR(128) NOT NULL	Authorization ID of the owner of the storage group.	G
VCATNAME	VARCHAR(24) NOT NULL	Name of the integrated catalog facility catalog.	G
	VARCHAR(24) NOT NULL	Not used	N
SPACE	INTEGER NOT NULL	Number of kilobytes of DASD storage allocated to the storage group as determined by the last execution of the STOSPACE utility.	G
	CHAR(5) NOT NULL	Not used	N
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies. RELCREATED should be used instead.	G
CREATEDBY	VARCHAR(128) NOT NULL WITH DEFAULT	Primary authorization ID of the user who created the storage group.	G
STATSTIME	TIMESTAMP NOT NULL WITH DEFAULT	If the STOSPACE utility was executed for the storage group, date and time when STOSPACE was last executed.	G
CREATEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the CREATE statement was executed for the storage group.	G
ALTEREDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the most recent ALTER STOGROUP statement was executed for the storage group. If no ALTER STOGROUP statement has been applied, ALTEREDTS has the value of CREATEDTS.	G
SPACEF	FLOAT NOT NULL WITH DEFAULT	Kilobytes of DASD storage for the storage group. The value is -1 if statistics have not been gathered. This is an updatable column.	G
DATACLAS	VARCHAR(24) NOT NULL	Name of the SMS data class. Blank if data class is not used.	G

Column name	Data type	Description	Use
MGMTCLAS	VARCHAR(24) NOT NULL	Name of the SMS management class. Blank if management class is not used.	G
STORCLAS	VARCHAR(24) NOT NULL	Name of the SMS storage class. Blank if storage class is not used.	G
CREATORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of creator: **blank** Authorization ID **L** Role	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. Blank if created prior to Version 9. See Release dependency indicators for all other values.	G

SYSIBM.SYSSTRINGS table

The SYSIBM.SYSSTRINGS table contains information about character conversion. Each row describes a conversion from one coded character set to another.

Also refer to z/OS C/C++ Programming Guide for information on the additional conversions that are supported.

Each row in the table must have a unique combination of values for its INCCSID, OUTCCSID, and IBMREQD columns. Rows for which the value of IBMREQD is N can be deleted, inserted, and updated subject to this uniqueness constraint and to the constraints imposed by a VALIDPROC defined on the table. An inserted row could have values for the INCCSID and OUTCCSID columns that match those of a row for which the value of IBMREQD is Y. DB2 then uses the information in the inserted row instead of the information in the IBM-supplied row. Rows for which the value of IBMREQD is Y cannot be deleted, inserted, or updated. For information about the use of inserted rows for character conversion, see *DB2 Installation Guide*.

DB2 has two methods for character conversions and applies them in the following order:

1. Conversions specified by the various combinations of the INCCSID and OUTCCSID columns in the SYSIBM.SYSSTRINGS catalog table.
2. Conversions provided by z/OS support for Unicode. For more information, see z/OS Support for Unicode: Using Conversion Services.

If neither of these methods can be used for a particular character conversion, DB2 returns an error.

Column name	Data type	Description	Use
INCCSID	INTEGER NOT NULL	The source CCSID for the character conversion represented by this row. The value of the source CCSID must be in the range of 1 to 65533 and must not be the same as the value for the OUTCCSID column.	G
OUTCCSID	INTEGER NOT NULL	The target CCSID for the character conversion represented by this row. The value of the target CCSID must be in the range of 1 to 65533 and must not be the same as the value for the INCCSID column.	G
TRANSTYPE	CHAR(2) NOT NULL	Indicates the nature of the conversion. Values can be: **GG** GRAPHIC to GRAPHIC **MM** EBCDIC MIXED to EBCDIC MIXED **MS** EBCDIC MIXED to SBCS **PM** ASCII MIXED to EBCDIC MIXED **PS** ASCII MIXED to SBCS **SM** SBCS to EBCDIC MIXED **SS** SBCS to SBCS **MP** EBCDIC MIXED to ASCII MIXED **PP** ASCII MIXED to ASCII MIXED **SP** SBCS to ASCII MIXED	G
ERRORBYTE	CHAR(1) FOR BIT DATA (Nulls are allowed)	The byte used in the conversion table as an error byte. Any non-null value that is specified for the ERRORBYTE column must not be the same as the value that is specified for the SUBBYTE column. Null indicates the absence of an error byte.	S

Column name	Data type	Description	Use
SUBBYTE	CHAR(1) FOR BIT DATA (Nulls are allowed)	The byte used in the conversion table as a substitution character. Any non-null value that is specified for the SUBBYTE column must not be the same as the value that is specified for the ERRORBYTE column. Null indicates the absence of a substitution character.	S
TRANSPROC	VARCHAR(24) NOT NULL WITH DEFAULT	The name of a module or blanks. A nonblank value must conform to the rules for z/OS program names. If IBMREQD is 'N', a nonblank value is the name of a conversion procedure provided by the user. The first five characters of the name of a user-provided conversion procedure must not be 'DSNXV'; these characters are used to distinguish user-provided conversion procedures from DB2 modules that contain DBCS conversion tables. If IBMREQD is 'Y', a nonblank value is the name of a DB2 module that contains DBCS conversion tables.	G

Column name	Data type	Description		Use
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape.		G
		Value	**Meaning**	
		B	Version 1R3 dependency indicator, not from the machine-readable material (MRM) tape	
		C	Version 2R1 dependency indicator, not from MRM tape	
		D	Version 2R2 dependency indicator, not from MRM tape	
		E	Version 2R3 dependency indicator, not from MRM tape	
		F	Version 3R1 dependency indicator, not from MRM tape	
		G	Version 4 dependency indicator, not from MRM tape	
		H	Version 5 dependency indicator, not from MRM tape	
		I	Version 6 dependency indicator, not from MRM tape	
		J	Version 6 dependency indicator, not from MRM tape	
		K	Version 7 dependency indicator, not from MRM tape	
		L	Version 8 dependency indicator, not from MRM tape	
		M	Version 9 dependency indicator, not from MRM tape	
		O	Version 10 dependency indicator, not from MRM tape	
		N	Not from MRM tape, no dependency	
		The value in this field is not a reliable indicator of release dependencies.		
TRANSTAB	VARCHAR(256) FOR BIT DATA NOT NULL WITH DEFAULT	Either a 256-byte conversion table or an empty (0 length) string.		S

SYSIBM.SYSSYNONYMS table

The SYSIBM.SYSSYNONYMS table contains one row for each synonym of a table or view.

Column name	Data type	Description	Use
NAME	VARCHAR(128) NOT NULL	Synonym for the table or view.	G
CREATOR	VARCHAR(128) NOT NULL	Authorization ID of the owner of the synonym.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table or view.	G
TBCREATOR	VARCHAR(128) NOT NULL	The schema of the table or view.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies. RELCREATED should be used instead.	G
CREATEDBY	VARCHAR(128) NOT NULL WITH DEFAULT	Primary authorization ID of the user who created the synonym.	G
CREATEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the CREATE statement was executed for the synonym. The value is '0001-01.01.00.00.00.000000' for synonyms created in a DB2 release prior to Version 5.	G
CREATORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of creator: **blank** Authorization ID **L** Role	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. Blank if created prior to Version 9. See Release dependency indicators for all other values.	G

SYSIBM.SYSTABAUTH table

The SYSIBM.SYSTABAUTH table records the privileges that users hold on tables and views.

Column name	Data type	Description	Use
GRANTOR	VARCHAR(128) NOT NULL	Authorization ID or role of the user who granted the privileges. Could also be PUBLIC.	G
GRANTEE	VARCHAR(128) NOT NULL	Authorization ID or role of the user who holds the privileges or the name of an application plan or package that uses the privileges. PUBLIC for a grant to PUBLIC.	G
GRANTEETYPE	CHAR(1) NOT NULL	Type of grantee: **blank** An authorization ID **L** Role **P** An application plan or a package. The grantee is a package if COLLID is not blank.	G
DBNAME	VARCHAR(24) NOT NULL	If the privileges were received from a user with DBADM, DBCTRL, or DBMAINT authority, DBNAME is the name of the database on which the GRANTOR has that authority. Otherwise, DBNAME is blank.	G
SCREATOR	VARCHAR(128) NOT NULL	If the row of SYSIBM.SYSTABAUTH was created as a result of a CREATE VIEW statement, SCREATOR is the schema of a table or view referred to in the CREATE VIEW statement. Otherwise, SCREATOR is the same as TCREATOR.	G
STNAME	VARCHAR(128) NOT NULL	If the row of SYSIBM.SYSTABAUTH was created as a result of a CREATE TABLE statement or a materialized query table, STNAME is the name of a table or view referred to in the fullselect of the CREATE TABLE statement.	G
TCREATOR	VARCHAR(128) NOT NULL	The schema of the table or view.	G
TTNAME	VARCHAR(128) NOT NULL	Name of the table or view.	G
AUTHHOWGOT	CHAR(1) NOT NULL WITH DEFAULT	Authorization level of the user from whom the privileges were received. This authorization level is not necessarily the highest authorization level of the grantor. **blank** Not applicable **B** System DBADM **C** DBCTRL **D** DBADM **E** SECADM **G** ACCESSCTRL **K** SQLADM **L** SYSCTRL **M** DBMAINT **S** SYSADM **T** DATAACCESS	G

46. PUBLIC followed by an asterisk (PUBLIC*) denotes PUBLIC AT ALL LOCATIONS.

Column name	Data type	Description	Use
	CHAR(12) NOT NULL	Internal use only	I
	CHAR(6) NOT NULL	Not used	N
	CHAR(8) NOT NULL	Not used	N
UPDATECOLS	CHAR(1) NOT NULL	The value of this column is blank if the value of UPDATEAUTH applies uniformly to all columns of the table or view. The value is an asterisk (*) if the value of UPDATEAUTH applies to some columns but not to others. In this case, rows will exist in SYSIBM.SYSCOLAUTH with matching timestamps and PRIVILEGE = blank. These rows list the columns on which update privileges have been granted.	G
ALTERAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can alter the table: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G
DELETEAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can delete rows from the table or view: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G
INDEXAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can create indexes on the table: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G
INSERTAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can insert rows into the table or view: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G
SELECTAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can select rows from the table or view: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G
UPDATEAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can update rows of the table or view: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

Column name	Data type	Description	Use
	VARCHAR(128) NOT NULL WITH DEFAULT	Not used	N
	VARCHAR(128) NOT NULL WITH DEFAULT	Not used	N
COLLID	VARCHAR(128) NOT NULL WITH DEFAULT	If the GRANTEE is a package, its collection name. Otherwise, the value is blank.	G
CONTOKEN	CHAR(8) NOT NULL WITH DEFAULT FOR BIT DATA	If the GRANTEE is a package, the consistency token of the DBRM from which the package was derived. Otherwise, the value is blank.	S
	CHAR(1) NOT NULL WITH DEFAULT	Not used	N
REFERENCESAUTH	CHAR(1) NOT NULL WITH DEFAULT	Whether the GRANTEE can create or drop referential constraints in which the table is a parent. **blank** Privilege is not held **G** Privilege held with the GRANT option **Y** Privilege held without the GRANT option	G
REFCOLS	CHAR(1) NOT NULL WITH DEFAULT	The value of this column is blank if the value of REFERENCESAUTH applies uniformly to all columns of the table. The value is an asterisk(*) if the value of REFERENCESAUTH applies to some columns but not to others. In this case, rows will exist in SYSIBM.SYSCOLAUTH with PRIVILEGE = R and matching timestamps that list the columns on which reference privileges have been granted.	G
GRANTEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the GRANT statement was executed.	G
TRIGGERAUTH	CHAR(1) NOT NULL WITH DEFAULT	Whether the GRANTEE can create triggers in which the table is named as the subject table: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G
GRANTORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of grantor: **blank** Authorization ID **L** Role	G

SYSIBM.SYSTABCONST table

The SYSIBM.SYSTABCONST table contains one row for each unique constraint (primary key or unique key) created in DB2 Version 7 or later.

Column name	Data type	Description	Use
CONSTNAME	VARCHAR(128) NOT NULL	Name of the constraint.	G
TBCREATOR	VARCHAR(128) NOT NULL	The schema of the table on which the constraint is defined.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table on which the constraint is defined.	G
CREATOR	VARCHAR(128) NOT NULL	Authorization ID under which the constraint was created.	G
TYPE	CHAR(1) NOT NULL	Type of constraint: **P** Primary key **U** Unique key	G
IXOWNER	VARCHAR(128) NOT NULL	The schema of the index enforcing the constraint or blank if index has not been created yet.	G
IXNAME	VARCHAR(128) NOT NULL	Name of the index enforcing the constraint or blank if index has not been created yet.	G
CREATEDTS	TIMESTAMP NOT NULL	Time when the statement to create the constraint was executed.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies. RELCREATED should be used instead.	G
COLCOUNT	SMALLINT NOT NULL	Number of columns in the constraint.	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. Blank if created prior to Version 9. See Release dependency indicators for all other values.	G

SYSIBM.SYSTABLEPART table

The SYSIBM.SYSTABLEPART table contains one row for each nonpartitioned table space and one row for each partition of a partitioned table space.

Column name	Data type	Description	Use
PARTITION	SMALLINT NOT NULL	Partition number; 0 if table space is not partitioned.	G
TSNAME	VARCHAR(24) NOT NULL	Name of the table space.	G
DBNAME	VARCHAR(24) NOT NULL	Name of the database that contains the table space.	G
IXNAME	VARCHAR(128) NOT NULL	Name of the partitioning index. This column is blank unless this is a table that uses index-controlled partitioning.	G
IXCREATOR	VARCHAR(128) NOT NULL	The schema of the partitioning index. This column is blank unless this is a table that uses index-controlled partitioning.	G
PQTY	INTEGER NOT NULL	For user-managed data sets, the value is the primary space allocation in units of 4 KB storage blocks or -1. PQTY is based on a value of PRIQTY in the appropriate CREATE or ALTER TABLESPACE statement. Unlike PQTY, however, PRIQTY asks for space in 1 KB units. A value of -1 indicates that either of the following cases is true: • PRIQTY was not specified for a CREATE TABLESPACE statement or for any subsequent ALTER TABLESPACE statements. • -1 was the most recently specified value for PRIQTY, either on the CREATE TABLESPACE statement or a subsequent ALTER TABLESPACE statement.	G
SQTY	SMALLINT NOT NULL	For user-managed data sets, the value is the secondary space allocation in units of 4 KB storage blocks or -1. SQTY is based on a value of SECQTY in the appropriate CREATE or ALTER TABLESPACE statement. Unlike SQTY, however, SECQTY asks for space in 1 KB units. A value of -1 indicates that either of the following cases is true: • SECQTY was not specified for a CREATE TABLESPACE statement or for any subsequent ALTER TABLESPACE statements. • -1 was the most recently specified value for SECQTY, either on the CREATE TABLESPACE statement or a subsequent ALTER TABLESPACE statement. If the value does not fit into the column, the value of the column is 32767. See the description of column SECQTYI.	G
STORTYPE	CHAR(1) NOT NULL	Type of storage allocation: **E** Explicit (storage group not used) **I** Implicit (storage group used)	G

Column name	Data type	Description	Use
STORNAME	VARCHAR(128) NOT NULL	Name of storage group used for space allocation. Blank if storage group not usedor for the catalog table spaces.	G
VCATNAME	VARCHAR(24) NOT NULL	Name of integrated catalog facility catalog used for space allocation.	G
CARD	INTEGER NOT NULL	Number of rows in the table space or partition or, if the table space is a LOB table space, the number of LOBs in the table space. The value is '2 147 483 647' if the number of rows is greater than or equal to '2 147 483 647'. The value is -1 if statistics have not been gathered.	G
FARINDREF	INTEGER NOT NULL	Number of rows that have been relocated far from their original page. The value is -1 if statistics have not been gathered. Not applicable if the table space is a LOB table space.	S
NEARINDREF	INTEGER NOT NULL	Number of rows that have been relocated near their original page. The value is -1 if statistics have not been gathered. Not applicable if the table space is a LOB table space.	S
PERCACTIVE	SMALLINT NOT NULL	Percentage of space occupied by rows of data from active tables. The value is -1 if statistics have not been gathered. The value is -2 if the table space is a LOB table space. This value is not applicable for understanding data distribution in tables that are organized for hash access.	S
PERCDROP	SMALLINT NOT NULL	Percentage of space occupied by rows of dropped tables. The value is -1 if statistics have not been gathered. The value is 0 for segmented table spaces. Not applicable if the table is an auxiliary table.	S
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies. RELCREATED should be used instead.	G
LIMITKEY	VARCHAR(765) NOT NULL	The high value of the partition in external format. If the table space was converted from index-controlled partitioning to table-controlled partitioning, the value is the highest possible value for an ascending key, or the lowest possible value for a descending key. If the table space is not partitioned, the value is an empty string. Beginning in Version 11, date and time values in the LIMITKEY field are delimited by single quotation marks (for example, '2001-01-01'). However, values that were added by previous versions of DB2 are not delimited. Therefore, this column can contain a mixture of delimited and non-delimited values.	G
FREEPAGE	SMALLINT NOT NULL	Number of pages loaded before a page is left as free space.	G
PCTFREE	SMALLINT NOT NULL	Percentage of each page left as free space.	G

Column name	Data type	Description	Use
CHECKFLAG	CHAR(1) NOT NULL WITH DEFAULT	**blank** The table space is not a partition, or does not contain rows that might violate referential constraints, check constraints, or both. **C** The table space partition is in a check pending status and there are rows in the table that can violate referential constraints, check constraints, or both. **D** The inline length of the LOB column that is associated with this LOB table space was decremented when the inline length was altered. **I** The inline length of the LOB column that is associated with this LOB table space was incremented when the inline length was altered.	G
	CHAR(4) NOT NULL WITH DEFAULT FOR BIT DATA	Not used	N
SPACE	INTEGER NOT NULL WITH DEFAULT	Number of kilobytes of DASD storage allocated to the table space partition, as determined by the last execution of the STOSPACE utility or RUNSTATS utility. **0** The STOSPACE or RUNSTATS utility has not been run. **-1** The table space was defined with the DEFINE NO clause, which defers the physical creation of the data sets until data is first inserted into one of the partitions, and data has yet to be inserted. **non-zero or non-negative value** An auxiliary table in the LOB table space. The value is if The value is updated by STOSPACE if the table space is related to a storage group. The value is updated by RUNSTATS if the utility is executed as RUNSTATS TABLESPACE with UPDATE(ALL) or UPDATE(SPACE).	G
COMPRESS	CHAR(1) NOT NULL WITH DEFAULT	Indicates the following: • For a table space partition, whether the COMPRESS attribute for the partition is YES. • For a nonpartitioned table space, whether the COMPRESS attribute is YES for the table space. Values for the column can be: **Y** Compression is defined for the table space **blank** No compression	G

Column name	Data type	Description	Use
PAGESAVE	SMALLINT NOT NULL WITH DEFAULT	Percentage of pages saved in the table space or partition as a result of defining the table space with COMPRESS YES. For example, a value of 25 indicates a savings of 25 percent, so that the pages required are only 75 percent of what would be required without data compression. The calculation includes overhead bytes for each row, the bytes required for dictionary, and the bytes required for the current FREEPAGE and PCTFREE specification for the table space or partition. This calculation is based on an average row length, and the result varies depending on the actual lengths of the rows. The value is 0 if there are no savings from using data compression, or if statistics have not been gathered. The value can be negative, if for example, data compression causes an increase in the number of pages in the data set.	S
STATSTIME	TIMESTAMP NOT NULL WITH DEFAULT	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.00.000000'.	G
GBPCACHE	CHAR(1) NOT NULL WITH DEFAULT	Group buffer pool cache option specified for this table space or table space partition. **A** Changed and unchanged pages are cached in the group buffer pool. **N** No data is cached in the group buffer pool. **S** Only changed system pages, such as space map pages that do not contain actual data values, are cached in the group buffer pool. **blank** Only changed pages are cached in the group buffer pool.	G
CHECKRID5B	CHAR(5) NOT NULL WITH DEFAULT FOR BIT DATA	Blank if the table or partition is not in a check pending status (CHECKFLAG is blank), or if the table space is not partitioned. Otherwise, the RID of the first row of the table space partition that can violate referential constraints, check constraints, or both; or the value is X'0000000000', indicating that any row can violate referential constraints.	S
TRACKMOD	CHAR(1) NOT NULL WITH DEFAULT	Whether to track the page modifications in the space map pages: **N** No **blank** Yes	G
EPOCH	INTEGER NOT NULL WITH DEFAULT	A number that increments whenever a utility operation that changes the location of rows in a table occurs.	G
SECQTYI	INTEGER NOT NULL WITH DEFAULT	Secondary space allocation in units of 4KB storage. For user-managed data sets, the value is the secondary space allocation in units of 4KB blocks.	G
CARDF	FLOAT NOT NULL WITH DEFAULT -1	Number of rows in the table space or partition, or if the table space is a LOB table space, the number of LOBs in the table space. The value is -1 if statistics have not been gathered.	G

Column name	Data type	Description	Use
IPREFIX	CHAR(1) NOT NULL WITH DEFAULT 'I'	The first character of the instance qualifier for the data set name for the table space or partition. 'I' or 'J' are the only valid characters for this field. The default is 'I'.	G
ALTEREDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the most recent ALTER TABLESPACE statement was executed for the table space or partition. If no ALTER TABLESPACE statement has been applied, the value is '0001-01-01.00.00.00.000000'.	G
SPACEF	FLOAT(8) NOT NULL WITH DEFAULT -1	Kilobytes of DASD storage. The value is -1 if statistics have not been gathered. The value might be non-zero for an auxiliary table in the LOB table space. This is an updatable column.	G
DSNUM	INTEGER NOT NULL WITH DEFAULT -1	Number of data sets. The value is -1 if statistics have not been gathered. This is an updatable column.	G
EXTENTS	INTEGER NOT NULL WITH DEFAULT -1	Number of data set extents. The value is -1 if statistics have not been gathered. This is an updatable column. This value is only for the last DSNUM for the object.	G
LOGICAL_PART	SMALLINT NOT NULL WITH DEFAULT	The logical partition (logical ascending or descending order) for table spaces created with either table-controlled partitioning or index-controlled partitioning. The physical partition number is kept in column PART and is zero for partitioned table spaces created prior to Version 8 and for nonpartitioned table spaces.	G
LIMITKEY_ INTERNAL	VARCHAR(512) NOT NULL WITH DEFAULT FOR BIT DATA	The highest value of the limit key of the partition in an internal format. If the uses index-controlled partitioning instead of table-controlled partitioning or the table is not partitioned, the value is an empty string. If the table space was converted from index-controlled partitioning to table-controlled partitioning, the value is the highest possible value for an ascending key, or the lowest possible value for a descending key. If any column of the key has a field procedure, the internal format is the encoded form of the value.	G
OLDEST_VERSION	SMALLINT NOT NULL WITH DEFAULT	The version number of the oldest format of data in the table part and any image copies at the part level.	G
CREATEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the partition was created.	G
AVGROWLEN	INTEGER NOT NULL WITH DEFAULT -1	Average length of rows for the tables in the table space or part. If the table space or part is compressed, the value is the compressed row length. If the table space or part is not compressed, the value is the uncompressed row length. The value is -1 if statistics have not been gathered.	G
FORMAT	CHAR(1) NOT NULL WITH DEFAULT	Indicates the format of the rows in the table space or partition: **R** Indicates reordered row format **blank** Indicates basic row format or a LOB table space	G

Column name	Data type	Description	Use
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. Blank if created prior to Version 9. See Release dependency indicators for all other values.	G
REORG_LR_TS	TIMESTAMP NOT NULL WITH DEFAULT	The time when the REORG or LOAD REPLACE utility last occurred. The default value is '0001-01-01.00.00.00.000000'.	G
HASHSPACE	BIGINT NOT NULL WITH DEFAULT	For partition-by-growth table spaces this is zero. For range-partitioned universal table spaces, this is the amount of space, in KB, specified at the partition level to override the space specification at the table level. If no override is provided it will be the same as the value of HASHSPACE in the SYSIBM.SYSTABLEPSACE catalog table.	G
HASHDATAPAGES	BIGINT NOT NULL WITH DEFAULT	For partition-by-growth table spaces, the value is zero. For range-partitioned universal table spaces, this is the number of hash data pages that correspond to the value of the HASHSPACE column for each partition. The value is 0 for table spaces which have been changed to use hash access but have not been reorganized.	G
RBA_FORMAT	CHAR(1) NOT NULL WITH DEFAULT	Indicates the format of the RBA/LRSN. **B** Basic, 6-byte RBA/LRSN format. **E** Extended, 10-byte RBA/LRSN format. **U** Undefined. DEFINE NO was specified when creating the table space, and the table space is not an XML table space with XML versions. **blank** For migrated objects.	G
PCTFREE_UPD	SMALLINT NOT NULL WITH DEFAULT	The percentage of free space that is reserved for updates to variable length records, as defined when the object as created or altered.	G
PCTFREE_UPD_CALC	SMALLINT NOT NULL WITH DEFAULT	The percentage of free space that is reserved for updates to variable length records, calculated by DB2 or utilities.	G

SYSIBM.SYSTABLEPART_HIST table

The SYSIBM.SYSTABLEPART_HIST table contains rows from SYSTABLEPART.

Rows are added or changed in this table when RUNSTATS collects history statistics. Rows in this table can also be inserted, updated, and deleted.

Column name	Data type	Description	Use
PARTITION	SMALLINT NOT NULL	Partition number. 0 if table space is not partitioned.	G
TSNAME	VARCHAR(24) NOT NULL	Name of the table space.	G
DBNAME	VARCHAR(24) NOT NULL	Name of the database that contains the table space.	G
PQTY	INTEGER NOT NULL	For user-managed data sets, the value is the primary space allocation in units of 4 KB storage blocks or -1. For user-specified values of PRIQTY other than -1, the value is set to the primary space allocation only if RUNSTATS TABLESPACE with UPDATE(ALL) or UPDATE(SPACE) is executed; otherwise, the value is zero. PQTY is based on a value of PRIQTY in the appropriate CREATE or ALTER TABLESPACE statement. Unlike PQTY, however, PRIQTY asks for space in 1 KB units. A value of -1 indicates that either of the following cases is true: • PRIQTY was not specified for a CREATE TABLESPACE statement or for any subsequent ALTER TABLESPACE statements. • -1 was the most recently specified value for PRIQTY, either on the CREATE TABLESPACE statement or a subsequent ALTER TABLESPACE statement. If a storage group is not used, the value is 0.	G

Column name	Data type	Description	Use
SECQTYI	INTEGER NOT NULL	For user-managed data sets, the value is the secondary space allocation in units of 4 KB storage blocks or -1. For user-specified values of SECQTY other than -1, the value is set to the secondary space allocation only if RUNSTATS TABLESPACE with UPDATE(ALL) or UPDATE(SPACE) is executed; otherwise, the value is zero. SQTY is based on a value of SECQTY in the appropriate CREATE or ALTER TABLESPACE statement. Unlike SQTY, however, SECQTY asks for space in 1 KB units. A value of -1 indicates that either of the following cases is true: • SECQTY was not specified for a CREATE TABLESPACE statement or for any subsequent ALTER TABLESPACE statements. • -1 was the most recently specified value for SECQTY, either on the CREATE TABLESPACE statement or a subsequent ALTER TABLESPACE statement. If a storage group is not used, the value is 0.	G
FARINDREF	INTEGER NOT NULL WITH DEFAULT -1	Number of rows that have been relocated far from their original page. The value is -1 if statistics have not been gathered. Not applicable if the table space is a LOB table space.	S
NEARINDREF	INTEGER NOT NULL WITH DEFAULT -1	Number of rows that have been relocated near their original page. The value is -1 if statistics have not been gathered. Not applicable if the table space is a LOB table space.	S
PERCACTIVE	SMALLINT NOT NULL WITH DEFAULT -1	Percentage of space occupied by rows of data from active tables. The value is -1 if statistics have not been gathered. The value is -2 if the table space is a LOB table space.	S
PERCDROP	SMALLINT NOT NULL WITH DEFAULT -1	Percentage of space occupied by rows of dropped tables. The value is -1 if statistics have not been gathered. The value is 0 for segmented table spaces. Not applicable if the table is an auxiliary table.	S
SPACEF	FLOAT(8) NOT NULL WITH DEFAULT -1	Number of kilobytes of DASD storage allocated to the table space partition. The value is -1 if statistics have not been gathered.	G

Column name	Data type	Description	Use
PAGESAVE	SMALLINT NOT NULL	Percentage of pages saved in the table space or partition as a result of defining the table space with COMPRESS YES. For example, a value of 25 indicates a savings of 25 percent, so that the pages required are only 75 percent of what would be required without data compression. The calculation includes overhead bytes for each row, the bytes required for dictionary, and the bytes required for the current FREEPAGE and PCTFREE specification for the table space or partition. This calculation is based on an average row length, and the result varies depending on the actual lengths of the rows. The value is 0 if there are no savings from using data compression, or if statistics have not been gathered. The value can be negative, if for example, data compression causes an increase in the number of pages in the data set.	S
STATSTIME	TIMESTAMP NOT NULL	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.00.000000'.	G
CARDF	FLOAT(8) NOT NULL WITH DEFAULT -1	Number of rows in the table space or partition, or if the table space is a LOB table space, the number of LOBS in the table space. The value is '-1' if statistics have not been gathered.	S
EXTENTS	INTEGER NOT NULL WITH DEFAULT -1	Number of data set extents. The value is '-1' if statistics have not been gathered. This value is only for the last DSNUM for the object.	G
DSNUM	INTEGER NOT NULL WITH DEFAULT -1	Data set number within the table space. For partitioned table spaces, this value corresponds to the partition number for a single partition copy, or 0 for a copy of an entire partitioned table space or index space. The value is '-1' if statistics have not been gathered.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
AVGROWLEN	INTEGER NOT NULL WITH DEFAULT -1	Average length of rows for the tables in the table space or part. If the table space or part is compressed, the value is the compressed row length. If the table space or part is not compressed, the value is the uncompressed row length. The value is '-1' if statistics have not been gathered.	G

SYSIBM.SYSTABLES table

The SYSIBM.SYSTABLES table contains one row for each table, view, or alias.

Column name	Data type	Description	Use
NAME	VARCHAR(128) NOT NULL	Name of the table, view, or alias.	G
CREATOR	VARCHAR(128) NOT NULL	The schema of the table, view, or alias.	G
TYPE	CHAR(1) NOT NULL	Type of object: **A** Alias **C** Clone table **G** Created global temporary table **H** History table **M** Materialized query table **P** Table that was implicitly created for XML columns **R** Archive table **T** Table **V** View **X** Auxiliary table	G
DBNAME	VARCHAR(24) NOT NULL	For a table, or a view of tables, the name of the database that contains the table space that is named in TSNAME. For a created temporary table, an alias, or a view of a view, the value is DSNDB06.	G
TSNAME	VARCHAR(24) NOT NULL	For a table, or a view of one table, the name of the table space that contains the table. For a view of more than one table, the name of a table space that contains one of the tables. For a created temporary table, a view of a view, or an alias, it is SYSTSTAB.	G
DBID	SMALLINT NOT NULL	Internal identifier of the database; 0 if the row describes a view, alias, or created temporary table. Non-zero if the view has an INSTEAD OF trigger defined.	S
OBID	SMALLINT NOT NULL	Internal identifier of the table; 0 if the row describes a view, an alias, or a created temporary table. Non-zero if the view has an INSTEAD OF trigger defined.	S
COLCOUNT	SMALLINT NOT NULL	Number of columns in the table or view. The value is 0 if the row describes an alias.	G
EDPROC	VARCHAR(24) NOT NULL	Name of the edit procedure; blank if the row describes a view or alias or a table without an edit procedure.	G
VALPROC	VARCHAR(24) NOT NULL	Name of the validation procedure; blank if the row describes a view or alias or a table without a validation procedure.	G
CLUSTERTYPE	CHAR(1) NOT NULL	Whether RESTRICT ON DROP applies: **blank** No **Y** Yes. You cannot drop the table or any table space or database that contains the table.	G
CLUSTERID	INTEGER NOT NULL	Not used	N

Column name	Data type	Description	Use
CARD	INTEGER NOT NULL	Not used	N
NPAGES	INTEGER NOT NULL	Total number of pages that include rows of the table. The value is -1 if statistics have not been gathered, or the row describes a view, an alias, a created temporary table, or an auxiliary table. This column can be updated.	S
PCTPAGES	SMALLINT NOT NULL	Percentage of active table space pages that contain rows of the table. A page is termed active if it is formatted for rows, regardless of whether it contains any. If the table space is segmented, the percentage is based on the number of active pages in the set of segments that are assigned to the table. The value is -1 if statistics have not been gathered, or the row describes a view, alias, created temporary table, or auxiliary table. This column can be updated.	S
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies. RELCREATED should be used instead.	G
REMARKS	VARCHAR(762) NOT NULL	A character string that is provided by the user with the COMMENT statement.	G
PARENTS	SMALLINT NOT NULL	Number of relationships in which the table is a dependent. The value is 0 if the row describes a view, an alias, a created temporary table, or a materialized query table.	G
CHILDREN	SMALLINT NOT NULL	Number of relationships in which the table is a parent. The value is 0 if the row describes a view, an alias, a created temporary table, or a materialized query table.	G
KEYCOLUMNS	SMALLINT NOT NULL	Number of columns in the primary key of the table. The value is 0 if the row describes a view, an alias, or a created temporary table.	G

Column name	Data type	Description	Use
RECLENGTH	SMALLINT NOT NULL	For user tables, the maximum length of any record in the table. Length is 8+*N*+*L*, where: • The number 8 accounts for the header (6 bytes) and the ID map entry (2 bytes). • *N* is 10 if the table has an edit procedure, or 0 otherwise. • *L* is the sum of the maximum column lengths. In determining the maximum length of a column, take into account whether the column allows nulls and the data type of the column. If the column can contain nulls and is not a LOB or ROWID column, add 1 byte for a null indicator. Use 4 bytes for the length of a LOB column and 19 bytes for the length of a ROWID column. If the column has a varying-length data type (for example, VARCHAR, CLOB, or BLOB), add 2 bytes for a length indicator. For more information about column lengths, see "Data types" on page 80. The value is 0 if the row describes a view, alias, or auxiliary table. For maximum row and record sizes, see Maximum record size.	G
STATUS	CHAR(1) NOT NULL	Indicates the status of the table definition: **I** The definition of the table is incomplete. The TABLESTATUS column indicates the reason why the table definition is incomplete. **R** An error occurred when an attempt was made to regenerate the internal representation of the view. **X** The table has a unique constraint (primary key or unique key) and the table definition is complete. **blank** The table has no unique constraint (primary key or unique key), the table is a catalog table, or the row describes a view or alias. The definition of the table, view, or alias is complete.	G
KEYOBID	SMALLINT NOT NULL	Internal DB2 identifier of the index that enforces uniqueness of the primary key of the table; 0 if not applicable.	S
LABEL	VARCHAR(90) NOT NULL	The label as given by a LABEL statement; otherwise, the value is an empty string.	G

Column name	Data type	Description	Use
CHECKFLAG	CHAR(1) NOT NULL WITH DEFAULT	**C** The table space that contains the table is in CHECK-pending status. One of the following conditions is true: • There are rows in the table that violate referential constraints, check constraints, or both • The table is a materialized query table that might contain inconsistent data **blank** Indicates one of the following conditions: • The table contains no rows that violate referential constraints, check constraints, or both • The table is a materialized query table that contains consistent data • The row describes a view, an alias, or a temporary table	G
CHECKRID	CHAR(4) NOT NULL WITH DEFAULT FOR BIT DATA	A value of 'FFFFFF00' in this column indicates that the edit procedure on this table is defined without row attribute sensitivity. Any other value indicates that the edit procedure is defined with row attribute sensitivity.	G
AUDITING	CHAR(1) NOT NULL WITH DEFAULT	Value of the audit option: **A** AUDIT ALL **C** AUDIT CHANGE **blank** AUDIT NONE, or the row describes a view, an alias, or a created temporary table.	G
CREATEDBY	VARCHAR(128) NOT NULL WITH DEFAULT	Primary authorization ID of the user who created the table, view, or alias.	G
LOCATION	VARCHAR(128) NOT NULL WITH DEFAULT	Location name of the object of an alias. The value is blank for a table, a view, an alias that was not defined with a three-part object name, or a materialized query table.	G
TBCREATOR	VARCHAR(128) NOT NULL WITH DEFAULT	• For an alias, the schema of the referenced table or view • For a base table that is involved in a clone relationship, the name of the creator of the clone table • For a clone table that is involved in a clone relationship, the name of the creator of the base table • For a view, the name of the underlying table. • Otherwise, TBCREATOR is blank	G
TBNAME	VARCHAR(128) NOT NULL WITH DEFAULT	• For an alias, the name for the referenced table or view • For a base table that is involved in a clone relationship, the name of the clone table • For a clone table that is involved in a clone relationship, the name of the base table • For a view, the name of the underlying table. • Otherwise, TBNAME is blank	G
CREATEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the CREATE statement was executed for the table, view, or alias	G

Column name	Data type	Description	Use
ALTEREDTS	TIMESTAMP NOT NULL WITH DEFAULT	For a table, the time when the latest ALTER TABLE statement was applied. If no ALTER TABLE statement was applied, or if the row is for an alias, ALTEREDTS has the value of CREATEDTS. For a view, the time when the last ALTER VIEW REGENERATE statement was applied.	G
DATACAPTURE	CHAR(1) NOT NULL WITH DEFAULT	Records the value of the DATA CAPTURE option for a table: **blank** No **Y** Yes For a created temporary table, DATACAPTURE is always blank.	G
RBA1	CHAR(10) NOT NULL WITH DEFAULT FOR BIT DATA	The log RBA when the table was created. Otherwise, RBA1 is X'00000000000000000000', indicating that the log RBA is not known, or that the object is a view, an alias, or a created temporary table. In a data sharing environment, RBA1 is the LRSN (Log Record Sequence Number) value.	S
RBA2	CHAR(10) NOT NULL WITH DEFAULT FOR BIT DATA	The log RBA when the table was last altered. Otherwise, RBA2 is X'00000000000000000000' indicating that the log RBA is not known, or that the object is a view, an alias, or a created temporary table. RBA1 equals RBA2 if the table has not been altered. In a data sharing environment, RBA2 is the LRSN (Log Record Sequence Number) value.	S
PCTROWCOMP	SMALLINT NOT NULL WITH DEFAULT	Percentage of rows that are compressed within the total number of active rows in the table. This number includes any row in a table space that is defined with COMPRESS YES. The value is -1 if statistics have not been gathered, or the row describes a view, alias, created temporary table, or auxiliary table. This column can be updated.	S
STATSTIME	TIMESTAMP NOT NULL WITH DEFAULT	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.00.000000'. For a created temporary table, the value of STATSTIME is always the default value. This column can be updated.	G
CHECKS	SMALLINT NOT NULL WITH DEFAULT	Number of check constraints that are defined on the table. The value is 0 if either of the following conditions are true: • The row describes a view, an alias, a created temporary table, or a materialized query table. • No constraints are defined on the table.	G
CARDF	FLOAT NOT NULL WITH DEFAULT -1	Total number of rows in the table or total number of LOBs in an auxiliary table. The value is -1 if statistics have not been gathered or the row describes a view, alias, or created temporary table. This column can be updated.	S

Column name	Data type	Description	Use
CHECKRID5B	CHAR(5) NOT NULL WITH DEFAULT FOR BIT DATA	RID of the first row of the table space partition that can violate referential constraints, check constraints, or both. The value of X'0000000000' indicates that any row can violate referential constraints. The value is blank if any of the following conditions are true: • The table or partition is not in CHECK-pending status (CHECKFLAG is blank) • The table space is not partitioned • The table is a created temporary table	S
ENCODING_SCHEME	CHAR(1) NOT NULL WITH DEFAULT 'E'	Encoding scheme for tables, views, and local aliases: **E** EBCDIC **A** ASCII **M** Multiple CCSID set or multiple encoding schemes **U** UNICODE **blank** For remote aliases The value is 'E' for tables in non-work file databases. The value is blank for tables in work file databases that were created before Version 5 or in the default database, DSNDB04. This column is not applicable for tables that were created before DB2 for z/OS Version 5.	G
TABLESTATUS	VARCHAR(30) NOT NULL WITH DEFAULT	Indicates the reason for an incomplete table definition: **L** Definition is incomplete because an auxiliary table or auxiliary index has not been defined for a LOB column. **P** Definition is incomplete because the table lacks a primary index. **R** Definition is incomplete because the table lacks a required index on a row ID. **U** Definition is incomplete because the table lacks a required index on a unique key. **V** An error occurred when an attempt was made to regenerate the internal representation of the view. **blank** Definition is complete.	G
NPAGESF	FLOAT(8) NOT NULL WITH DEFAULT -1	Number of pages that are used by the table. The value is -1 if statistics have not been gathered or the table is an auxiliary table. This column can be updated.	G
SPACEF	FLOAT(8) NOT NULL WITH DEFAULT -1	Kilobytes of DASD storage. The value is -1 if statistics have not been gathered. The value might be non-zero for an auxiliary table in the LOB table space. This column can be updated.	G
AVGROWLEN	INTEGER NOT NULL WITH DEFAULT -1	Average length of rows for the tables in the table space. If the table space is compressed, the value is the compressed row length. If the table space is not compressed, the value is the uncompressed row length. The value is -1 if statistics have not been gathered.	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. See Release dependency indicators for the values.	G

Column name	Data type	Description	Use
NUM_DEP_MQTS	SMALLINT NOT NULL WITH DEFAULT	Number of dependent materialized query tables. The value is zero if the row describes an alias or a created temporary table, or if no materialized query tables are defined on the table.	G
VERSION	SMALLINT NOT NULL WITH DEFAULT	The version of the data row format for this table. • A value of zero indicates that an alter operation that creates a new version has never occurred for this table. • A value of -1 indicates that the view has been regenerated because a column of the base table has been altered. • A value of 800 indicates that a successful CREATE VIEW or ALTER VIEW statement has occurred against this view in Version 8 or later. • A value of 900 indicates that a successful ALTER TABLE statement with a DROP COLUMN clause has occurred against this view.	G
PARTKEYCOLNUM	SMALLINT NOT NULL WITH DEFAULT	The number of columns in the partitioning key. This value is zero for tables that do not have partitioning or use index-controlled partitioning. The value is non-zero for tables that use table-controlled partitioning.	G
SPLIT_ROWS	CHAR(1) NOT NULL WITH DEFAULT	This column is blank except for volatile tables. For volatile table, this column contains 'Y' to indicate to DB2 to use index access on this table whenever possible.	G
SECURITY_LABEL	CHAR(1) NOT NULL	This column is only meaningful if the TYPE column is a T (for table) or M (for materialized query table). The value indicates whether the table has multilevel security: **Blank** The table does not have multilevel security. **R** The table has multilevel security with row granularity.	G
OWNER	VARCHAR(128) NOT NULL WITH DEFAULT	Authorization ID of the owner of the table, view, or alias. This column is blank for tables, views, or aliases that were created before DB2 for z/OSVersion 9.	G
APPEND	CHAR(1) NOT NULL WITH DEFAULT	Indicates whether the APPEND option is specified for the table. **Y** The APPEND option is specified. **N** The APPEND option is not specified.	G
OWNERTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of owner: **blank** Authorization ID **L** Role	G
CONTROL	CHAR(1) NOT NULL WITH DEFAULT	Indicates whether access to the table is enforced by using row or column access control: **blank** No access control enforcement **B** The table is enforced by using both row and column access control **C** The table is enforced by using column access control **R** The table is enforced by using row access control	G

Column name	Data type	Description	Use
VERSIONING _SCHEMA	VARCHAR(128) NOT NULL WITH DEFAULT	Indicates the schema name of the history table if the table is a system-period temporal table with versioning. Indicates the schema name of the system-period temporal table if the table is a history table. Otherwise, the value is blank.	G
VERSIONING _TABLE	VARCHAR(128) NOT NULL WITH DEFAULT	Indicates the table name of the history table if the table is a system-period temporal table with versioning. Indicates the table name of system-period temporal table if the table is a history table. Otherwise, the value is blank.	G
HASHKEYCOLUMNS	SMALLINT NOT NULL WITH DEFAULT	The number of columns in the hash key of the table. The value is 0 if the row describes a view, an alias, or a created temporary table.	G
ARCHIVING_ SCHEMA	VARCHAR(128) NOT NULL WITH DEFAULT	Contains a schema name as follows: • If the table is an archive-enabled table, this column contains the schema name of the archive table. • If the table is an archive table, this column contains the schema name of the archive-enabled table. • If the table is not an archive-enabled table or an archive table, the value is blank.	G
ARCHIVING _TABLE	VARCHAR(128) NOT NULL WITH DEFAULT	Contains a table name as follows: • If the table is an archive-enabled table, this column contains the table name of the archive table. • If the table is an archive table, this column contains the table name of the archive-enabled table. • If the table is not an archive-enabled table or an archive table, the value is blank.	G
STATS_FEEDBACK	CHAR (1) NOT NULL	When a query qualifies for statistics collection based on DSNZPARM STATFDBK_SCOPE, this column controls whether statistics recommendations for this table are placed in SYSIBM.SYSSTATFEEDBACK. You can update this flag to 'Y' or 'N' to enable or disable collection for the table.	G

SYSIBM.SYSTABLESPACE table

The SYSIBM.SYSTABLESPACE table contains one row for each table space.

Column name	Data type	Description	Use
NAME	VARCHAR(24) NOT NULL	Name of the table space.	G
CREATOR	VARCHAR(128) NOT NULL	Authorization ID of the owner of the table space.	G
DBNAME	VARCHAR(24) NOT NULL	Name of the database that contains the table space.	G
DBID	SMALLINT NOT NULL	Internal identifier of the database which contains the table space.	S
OBID	SMALLINT NOT NULL	Internal identifier of the table space file descriptor.	S
PSID	SMALLINT NOT NULL	Internal identifier of the table space page set descriptor.	S
BPOOL	CHAR(8) NOT NULL	Name of the buffer pool used for the table space.	G
PARTITIONS	SMALLINT NOT NULL	Number of partitions of the table space; 0 if the table space is not partitioned.	G
LOCKRULE	CHAR(1) NOT NULL	Lock size of the table space: **A** Any **L** Large object (LOB) **P** Page **R** Row **S** Table space **T** Table **X** implicitly created XML table space	G
PGSIZE	SMALLINT NOT NULL	Size of pages in the table space in kilobytes.	G
ERASERULE	CHAR(1) NOT NULL	Whether the data sets are to be erased when dropped. The value is meaningless if the table space is partitioned. **N** No erase **Y** Erase	G
STATUS	CHAR(1) NOT NULL	Availability status of the table space: **A** Available **C** Definition is incomplete because the table space does not use table-controlled partitioning and a partitioning index has not been created. **P** Table space is in a check pending status. **S** Table space is in a check pending status with the scope less than the entire table space. **T** Definition is incomplete because no table has been created.	G

Column name	Data type	Description	Use
IMPLICIT	CHAR(1) NOT NULL	Whether the table space was created implicitly: **N** No **Y** Yes	G
NTABLES	SMALLINT NOT NULL	Number of tables defined in the table space.	G
NACTIVE	INTEGER NOT NULL	Number of active pages in the table space. A page is termed active if it is formatted for rows, even if it currently contains none. The value is 0 if statistics have not been gathered. This is an updatable column.	S
	VARCHAR(24) NOT NULL	Not used	N
CLOSERULE	CHAR(1) NOT NULL	Whether the data sets are candidates for closure when the limit on the number of open data sets is reached. **N** No **Y** Yes	G
SPACE	INTEGER NOT NULL	Number of kilobytes of DASD storage allocated to the table space, as determined by the last execution of the STOSPACE utility. The value is 0 if the table space is not related to a storage group, or if STOSPACE has not been run. If the table space is partitioned, the value is the total kilobytes of DASD storage allocated to all partitions that are storage group defined.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. If ALTER TABLESPACE changes the DSSIZE value to 128G or 256G, this column value is changed to O, which is the release dependency indicator for Version 10. The value in this field is not a reliable indicator of release dependencies. RELCREATED should be used instead.	G
	VARCHAR(54) NOT NULL	Internal use only	I
	VARCHAR(24) NOT NULL	Internal use only	I
SEGSIZE	SMALLINT NOT NULL WITH DEFAULT	Number of pages in each segment of a segmented table space. The value is 0 if the table space is not segmented.	G
CREATEDBY	VARCHAR(128) NOT NULL WITH DEFAULT	Primary authorization ID of the user who created the table space.	G
STATSTIME	TIMESTAMP NOT NULL WITH DEFAULT	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.00.000000'. This is an updatable column.	G

Column name	Data type	Description	Use
LOCKMAX	INTEGER	The maximum number of locks per user to acquire for the table or table space before escalating to the next locking level. **0** Lock escalation does not occur. *n* *n*, where *n* > 0, is the maximum number of locks (row, page, or LOB locks for the table or table space) an application process can acquire before lock escalation occurs. **-1** Represents LOCKMAX SYSTEM. The value of field LOCKS PER TABLE(SPACE) on installation panel DSNTIPJ determines lock escalation. If the value of the field is 0, lock escalation does not occur. If the value is *n*, where *n* > 0, lock escalation occurs as it does for LOCKMAX n.	G
TYPE	CHAR(1) NOT NULL WITH DEFAULT	The type of table space: **blank** The table space was created without the LOB or MEMBER CLUSTER options. If the DSSIZE column is zero, the table space is not greater than 64 gigabytes. **G** The table space was defined with the MAXPARTITIONS option (a partitioned-by-growth table space) with the underlying structure of a universal table space. **L** The table space can be greater than 64 gigabytes. **O** The table space was defined with the LOB option (the table space is a LOB table space). **P** Implicit table space created for XML columns. **R** Range-partitioned universal table space.	G
CREATEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the CREATE statement was executed for the table space. If the table space was created in a DB2 release prior to Version 5, the value is '0001-01-01.00.00.00.000000'.	G
ALTEREDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the most recent ALTER TABLESPACE statement was executed for the table space. If no ALTER TABLESPACE statement has been applied, ALTEREDTS has the value of CREATEDTS. If the index was created in a DB2 release prior to Version 5, the value is '0001-01-01.00.00.00.000000'.	G
ENCODING_SCHEME	CHAR(1) NOT NULL WITH DEFAULT 'E'	Default encoding scheme for the table space: **E** EBCDIC **A** ASCII **U** UNICODE **blank** For table spaces in a work file database or a TEMP database (a database that was created AS TEMP, which is for declared temporary tables.) The value is 'E' for tables in non work file databases and blank for tables in work file databases created prior to Version 5 or the default database, DSNDB04.	G
SBCS_CCSID	INTEGER NOT NULL WITH DEFAULT	Default SBCS CCSID for the table space. For a table space in a work file database, a TEMP database, or a database created in a DB2 release prior to Version 5, the value is 0.	G
DBCS_CCSID	INTEGER NOT NULL WITH DEFAULT	Default DBCS CCSID for the table space. For a table space in a work file database, a TEMP database, or a database created in a DB2 release prior to Version 5, the value is 0.	G

Column name	Data type	Description	Use
MIXED_CCSID	INTEGER NOT NULL WITH DEFAULT	Default mixed CCSID for the table space. For a table space in a work file database, a TEMP database, or a database created in a DB2 release prior to Version 5, the value is 0.	G
MAXROWS	SMALLINT NOT NULL DEFAULT 255	The maximum number of rows that DB2 will place on a data page. The default value is 255. For a LOB table space, the value is 0 to indicate that the column is not applicable.	G
	CHAR(1) NOT NULL WITH DEFAULT	Not used	N
LOG	CHAR(1) NOT NULL WITH DEFAULT 'Y'	Whether the changes to a table space are to be logged. **N** This table space has the NOT LOGGED attribute. Undo and redo logging for the table space and all indexes for tables in the table space is suppressed. Logging is also suppressed for the auxiliary indexes for all auxiliary tables associated with tables in the table space. **Y** This table space has the LOGGED attribute. Normal logging is associated with modifications to this table space, all indexes for tables in this table space, and all auxiliary indexes for all auxiliary tables associated with tables in the table space. **X** This LOB or XML table space has the NOT LOGGED attribute. Undo and redo logging for the table space is suppressed. Also, the logging attribute for this LOB or XML table space is linked to the logging attribute of the associated base table space and might not be able to be altered independently. If the logging attribute of the base table space is altered to LOGGED, the logging attribute of the LOB or XML table space will also be altered to LOGGED.	G
NACTIVEF	FLOAT NOT NULL WITH DEFAULT -1	Number of active pages in the table space. A page is termed active if it is formatted for rows, even if it currently contains none. The value is -1 if statistics have not been gathered. This is an updatable column.	S
DSSIZE	INTEGER NOT NULL WITH DEFAULT	Maximum size of a data set in kilobytes. The value might be 0 if the table space was created prior to Version 10, but will contain the actual value after the table space is converted to a partitioned by growth table space.	G
OLDEST_VERSION	SMALLINT NOT NULL WITH DEFAULT	The version number of the oldest format of data in the table space and any image copies.	G
CURRENT_VERSION	SMALLINT NOT NULL WITH DEFAULT	The version number describing the newest format of data in the table space. A zero indicates that the table space has never had versioning. After the version number reaches the maximum value, the number wraps back to one.	G

Column name	Data type	Description	Use
AVGROWLEN	INTEGER NOT NULL WITH DEFAULT -1	Average length of rows for the tables in the table space or part. If the table space or part is compressed, the value is the compressed row length. If the table space or part is not compressed, the value is the uncompressed row length. The value is -1 if statistics have not been gathered.	G
SPACEF	FLOAT NOT NULL WITH DEFAULT	Kilobytes of DASD storage for the storage group. The value is -1 if statistics have not been gathered. This is an updatable column.	G
CREATORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of creator: **blank** Authorization ID **L** Role	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. Blank if created prior to Version 9. See Release dependency indicators for all other values.	G
INSTANCE	SMALLINT NOT NULL WITH DEFAULT	INSTANCE indicates the data set instance number of the current base object (table and index).	G
CLONE	CHAR(1) NOT NULL WITH DEFAULT	Indicates whether the table space contains any objects that are involved in a clone relationship: **Y** Table space contains objects that are involved in a clone relationship **N** Table space does not contain any objects that are involved in a clone relationship	G
MAXPARTITIONS	SMALLINT NOT NULL WITH DEFAULT	Identifies the maximum number of partitions to which the table space can grow. 0 if the table space is not a partition-by-growth table space.	G
MEMBER_CLUSTER	CHAR(1) NOT NULL WITH DEFAULT	Whether MEMBER CLUSTER is specified for the table space: **Y** MEMBER CLUSTER is specified for the table space **blank** MEMBER CLUSTER is not specified for the table space	G
ORGANIZATIONTYPE	CHAR(1) NOT NULL WITH DEFAULT	Type of table space organization: **blank** Not known. Blank is the default. **H** Hash organization	G
HASHSPACE	BIGINT NOT NULL WITH DEFAULT	The amount of space, in KB, that is to be allocated to the table space or partition as hash space. For partition-by-growth table spaces, the space applies to the whole table space. For range-partitioned universal table spaces, the space is applicable for each partition.	G

Column name	Data type	Description	Use
HASHDATAPAGES	BIGINT NOT NULL WITH DEFAULT	The total number of hash data pages to preallocate for hash space. For partition-by-growth table spaces, this includes all pages in the fixed part of the table space. For range-partitioned universal table spaces, this is the number of pages in the fixed hash space in each partition unless it is overridden by providing hash space at the partition level. This is calculated by DB2 from the value specified with the HASH SPACE option or when the REORG utility is run with automatic estimation of space. The calculated value is used in the hash algorithm. The value is 0 for non-hash table spaces. The value is also 0 for table spaces which have been changed to use hash access but have not been reorganized.	G

SYSIBM.SYSTABLESPACESTATS table

The SYSIBM.SYSTABLESPACESTATS table contains real time statistics for table spaces.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
UPDATESTATSTIME	TIMESTAMP NOT NULL WITH DEFAULT	The timestamp that the row in the SYSTABLESPACESTATS table is inserted or updated.	G
NACTIVE	INTEGER	The number of active pages in the table space or partition.	G
NPAGES	INTEGER	The number of distinct pages with active rows in the partition or table space. This is an updatable column. This column can be used to calculate an estimate of the size of LOB data in a table space. To produce an estimate, use the following formula: *value of NPAGES * page size* = approximate size of LOB data	G
EXTENTS	SMALLINT	The number of extents in the table space. For multi-piece table spaces, this value is the number of extents for the last data set. For a data set that is striped across multiple volumes, the value is the number of logical extents. A null value indicates the number of extents is unknown.	G
LOADRLASTTIME	TIMESTAMP	The timestamp that the LOAD REPLACE utility was last run on the table space or partition. A null value indicates that the LOAD REPLACE utility has never been run on the table space or partition or that the timestamp is unknown.	G
REORGLASTTIME	TIMESTAMP	The timestamp the REORG utility was last run on the table space or partition, or when the REORG utility has not been run, the time when the table space or partition was created. A null value indicates that the timestamp is unknown.	G
REORGINSERTS	INTEGER	The number of records or LOBs that have been inserted into the table space or partition or loaded into the table space or partition using the LOAD utility specified without the REPLACE option since the last time the REORG or LOAD REPLACE utilities were run, or since the object was created. A null value indicates that the number of inserted records or LOBs is unknown.	G
REORGDELETES	INTEGER	The number of records or LOBs that have been deleted from the table space or partition since the last time the REORG or LOAD REPLACE utilities were run, or since the object was created. A null value indicates that the number of deleted records or LOBs is unknown.	G

Column name	Data type	Description	Use
REORGUPDATES	INTEGER	The number of rows that have been updated in the table space or partition since the last time the REORG or LOAD REPLACE utilities were run, or since the object was created. A null value indicates that the number of updated rows is unknown.	G
REORGUNCLUSTINS	INTEGER	The number of records that were inserted that are not well-clustered with respect to the clustering index since the last REORG or LOAD REPLACE , or since the object was created. A record is well-clustered if the record is inserted into a page that is within 16 pages of the ideal candidate page. The clustering index determines the ideal candidate page. A null value indicates that the number of not well clustered pages is unknown.	G
REORGDISORGLOB	INTEGER	The number of LOBs that were inserted that are not perfectly chunked since the last REORG or LOAD REPLACE, or since the object was created. A LOB is perfectly chunked if the allocated pages are in the minimum number of chunks. A null value indicates that the number of not perfectly chunked LOBs is unknown.	G
REORGMASSDELETE	INTEGER	The number of mass deletes from a segmented or LOB table space, or the number of dropped tables from a segmented table space since the last time the REORG or LOAD REPLACE utilities were run, or since the object was created. A null value indicates that the number of mass deletes is unknown.	G
REORGNEARINDREF	INTEGER	The number of overflow records that are created and relocated near the pointer record since the last time the REORG and LOAD REPLACE utilities were run, or since the object was created. For non-segmented table spaces, a page is near the present page if the two page numbers differ by 16 or less. For segmented table spaces, a page is near the present page if the two page numbers differ by SEGSIZE*2 or less. A null value indicates that the number of overflow records that are near the pointer record is unknown.	G
REORGFARINDREF	INTEGER	The number of overflow records that are created and relocated far from the pointer record since the last time the REORG and LOAD REPLACE utilities were run, or since the object was created. For non-segmented table spaces, a page is far from the present page if the two page numbers differ by more than 16. For segmented table spaces, a page is far from the present page if the two page numbers differ by at least (SEGSIZE*2)+1. A null value indicates that the number of overflow records that are near the pointer record is unknown.	G
STATSLASTTIME	TIMESTAMP	The timestamp of the last time that the RUNSTATS utility is run on the table space or partition.	G

Column name	Data type	Description	Use
STATSINSERTS	INTEGER	The number of records or LOBs that have been inserted into the table space or partition or loaded into the table space or partition using the LOAD utility specified without the REPLACE option since the last time that the RUNSTATS utility was run, or since the object was created. A null value indicates that the number of inserted records or LOBs is unknown.	G
STATSDELETES	INTEGER	The number of records or LOBs that have been deleted from the table space or partition since the last time that the RUNSTATS utility was run, or since the object was created. A null value indicates that the number of deleted records or LOBs is unknown.	G
STATSUPDATES	INTEGER	The number of rows that have been updated in the table space or partition since the last time that the RUNSTATS utility was run, or since the object was created. A null value indicates that the number of updated rows is unknown.	G
STATSMASSDELETE	INTEGER	The number of mass deletes from a segmented or LOB table space, or the number of tables that are dropped from a segmented table space, since the last time the RUNSTATS utility was run, or since the object was created. A null value indicates that the number of mass deletes is unknown.	G
COPYLASTTIME	TIMESTAMP	The timestamp of the last full or incremental image copy of the table space or partition, or since the object was created. A null value indicates that the COPY utility has never been run on the table space or partition. A null value can also indicate that the timestamp of the last image copy is unknown.	G
COPYUPDATED-PAGES	INTEGER	The number of distinct pages that have been updated since the last time that the COPY utility was run, or since the object was created. A null value indicates that the number of updated pages is unknown.	G
COPYCHANGES	INTEGER	The number of insert, update, and delete operations, or the number of records loaded, since the last time that the COPY utility was run, or since the object was created. A null value indicates that the number of insert, update, and delete operations or the number of records loaded is unknown.	G
COPYUPDATELRSN	CHAR(10) FOR BIT DATA	The LRSN or RBA of the first update that occurs after the last time the COPY utility was run, or since the object was created. A null value indicates that the LRSN or RBA is unknown.	G

Column name	Data type	Description	Use
COPYUPDATETIME	TIMESTAMP	The timestamp of the first update that occurs after the last time that the COPY utility was run, or since the object was created. A null value indicates that the timestamp is unknown. The value is 6 bytes of X'FF' if the RBA/LRSN exceeds the 6-byte limit.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
DBID	SMALLINT NOT NULL	The internal identifier of the database. This column is used to map a DBID to its statistics.	G
PSID	SMALLINT NOT NULL	The internal identifier of the table space page set descriptor. This column is used to map a PSID to its statistics.	G
PARTITION	SMALLINT NOT NULL	The data set number within the table space. This column is used to map a data set number in a table space to its statistics. For partitioned table spaces, this value corresponds to the partition number for a single partition. For non-partitioned table spaces, this value is 0.	G
INSTANCE	SMALLINT NOT NULL WITH DEFAULT 1	Indicates if the object is associated with data set instance 1 or 2. This is an updatable column.	G
SPACE	BIGINT	The amount of space, in KB, that is allocated to the table space or partition. For multi-piece, linear page sets, this value is the amount of space in all data sets. A null value indicates the amount of space is unknown.	G
TOTALROWS	BIGINT	The number of rows or LOBs that are in the table space or partition. For XML, this column contains the number of physical records in the table space or partition. Each XML document might have more than one physical record in a table space or partition.	G
DATASIZE	BIGINT	The total number of bytes that row data occupy. For LOB table spaces this column is always 0. This is an updatable column.	G
UNCOMPRESSED-DATASIZE	BIGINT	This column is not used. The value is always set to 0.	G
DBNAME	VARCHAR(24) NOT NULL	The name of the database. This column is used to map a database to its statistics.	G
NAME	VARCHAR(24) NOT NULL	The name of the table space. This column is used to map a table space to its statistics.	G

Column name	Data type	Description	Use
REORGSCAN-ACCESS	BIGINT	The number of times data is accessed for SELECT, FETCH, searched UPDATE, or searched DELETE since the last CREATE, LOAD REPLACE or REORG, or since the object was created.A null value indicates that the number of times data is accessed is unknown.	G
REORGHASH-ACCESS	BIGINT	The number of times data is accessed using hash access for SELECT, FETCH, searched UPDATE, searched DELETE, or used to enforce referential integrity constraints since the last CREATE, LOAD REPLACE or REORG, or since the object was created. A null value indicates that the number of times data is accessed is unknown.	G
HASHLASTUSED	TIMESTAMP	The date when hash access was last used for SELECT, FETCH, searched UPDATE, searched DELETE, or used to enforce referential integrity constraints.	G
REORGCLUSTERSENS	BIGINT	The number of times data has been read by SQL statements that are sensitive to the clustering sequence of the data since the last REORG or LOAD REPLACE, or since the object was created.	G
DRIVETYPE	CHAR(3) NOT NULL WITH DEFAULT	The drive type on which the table space or table space partition data set is defined. **HDD** Hard Disk Drive **SSD** Solid State Drive For multi-volume data sets, the drive type is set to SSD if any volume is SSD. For multi-piece linear page sets, the drive type of the first data set is used. This is an updatable column.	G
LPFACILITY	CHAR(1)	Whether the disk control unit has the high performance list prefetch facility. **N** No **Y** Yes A NULL value indicates that it is unknown whether the disk control unit has the high performance list prefetch facility. This is an updatable column.	G
	BIGINT	Reserved for future IBM use.	R
UPDATESIZE	BIGINT	The net number of bytes that were added or removed by UPDATE operations since the object was created, or since the last REORG or LOAD REPLACE operation.	G
LASTDATACHANGE	TIMESTAMP	The last time that this row was updated because data was modified in the table space or partition. The timestamp reflects the time at which the real-time statistics table was updated, and not the time at which the data in the table space or partition was modified. Physical data changes such as reorganization of data are not reflected in this column.	G

In data sharing environments, the values in SYSIBM.SYSTABLESPACESTATS can be negative for short periods of time for certain situations.

Related concepts:

How DB2 maintains in-memory statistics in data sharing (DB2 Data Sharing Planning and Administration)

SYSIBM.SYSTABLES_HIST table

The SYSIBM.SYSTABLES_HIST table contains rows from SYSTABLES.

Rows are added or changed in this table when RUNSTATS collects history statistics. Rows in this table can also be inserted, updated, and deleted.

Column name	Data type	Description	Use
NAME	VARCHAR(128) NOT NULL	Name of the table, view, or alias.	G
CREATOR	VARCHAR(128) NOT NULL	The schema of the table, view, or alias.	G
DBNAME	VARCHAR(24) NOT NULL	For a table, or a view of tables, the name of the database that contains the table space named in TSNAME. For a temporary table, an alias, or a view of a view, the value is DSNDB06.	G
TSNAME	VARCHAR(24) NOT NULL	For a table, or a view of one table, the name of the table space that contains the table. For a view of more than one table, the name of a table space that contains one of the tables. For a temporary table, a view of a view, or an alias, it is SYSTSTAB.	G
COLCOUNT	SMALLINT NOT NULL	Number of columns in the table or view. The value is 0 if the row describes an alias.	G
PCTPAGES	SMALLINT NOT NULL WITH DEFAULT -1	Percentage of active table space pages that contain rows of the table. A page is termed active if it is formatted for rows, regardless of whether it contains any. If the table space is segmented, the percentage is based on the number of active pages in the set of segments assigned to the table. The value is -1 if statistics have not been gathered, or the row describes a view, alias, temporary table, or auxiliary table.	S
PCTROWCOMP	SMALLINT NOT NULL WITH DEFAULT -1	Percentage of rows compressed within the total number of active rows in the table. This includes any row in a table space that is defined with COMPRESS YES. The value is -1 if statistics have not been gathered, or the row describes a view, alias, temporary table, or auxiliary table.	G
STATSTIME	TIMESTAMP NOT NULL	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.00.000000'. For a temporary table, the value of STATSTIME is always the default value.	G
CARDF	FLOAT(8) NOT NULL WITH DEFAULT -1	Total number of rows in the table or total number of LOBs in an auxiliary table. The value is -1 if statistics have not been gathered or the row describes a view, alias, or temporary table.	S
NPAGESF	FLOAT(8) NOT NULL WITH DEFAULT -1	Total number of pages on which rows of the partition appear. The value is -1 if statistics have not been gathered.	S

Column name	Data type	Description	Use
AVGROWLEN	INTEGER NOT NULL WITH DEFAULT -1	Average row length of the table specified in the table space. The value is -1 if statistics have not been gathered.	G
SPACEF	FLOAT(8) NOT NULL WITH DEFAULT -1	Kilobytes of DASD storage. The value is -1 if statistics have not been gathered. This is an updatable column.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

SYSIBM.SYSTABLES_PROFILES table

The SYSIBM.SYSTABLES_PROFILES table contains one row for each profile that is associated with a table in SYSIBM.SYSTABLES.

Column name	Data type	Description	Use
SCHEMA	VARCHAR(128) NOT NULL	The schema (qualifier) for the table.	G
TBNAME	VARCHAR(128) NOT NULL	The table name.	G
PROFILE_TYPE	VARCHAR(32) NOT NULL	The type of profile. Allowed values are 'RUNSTATS'.	G
	VARCHAR(32)	Internal use only.	I
PROFILE_TEXT	CLOB(1M)	The text of the profile.	G
ROWID	ROWID NOT NULL GENERATED ALWAYS	The ROWID value for the LOB column of this table.	G
PROFILE_UPDATE	TIMESTAMP NOT NULL	The last time the profile was updated, or the timestamp for when the profile was inserted into the table.	G
	TIMESTAMP	Internal use only.	I

Related concepts:

RUNSTATS profiles (DB2 Utilities)

Related tasks:

Automating statistics maintenance (DB2 Performance)

SYSIBM.SYSTABLES_PROFILE_TEXT table

The SYSIBM.SYSTABLES_PROFILE_TEXT table is an auxiliary table for the PROFILE_TEXT column of the SYSIBM.SYSTABLES_PROFILES table and is required to hold LOB data.

Column name	Data type	Description	Use
PROFILE_TEXT	CLOB(2M) NOT NULL WITH DEFAULT	The complete text for the profile that the row represents.	G

SYSIBM.SYSTABSTATS table

The SYSIBM.SYSTABSTATS table contains one row for each partition of a partitioned table space.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
CARD	INTEGER NOT NULL	Total number of rows in the partition.	S
NPAGES	INTEGER NOT NULL	Total number of pages on which rows of the partition appear.	S
PCTPAGES	SMALLINT NOT NULL	Percentage of total active pages in the partition that contain rows of the table.	S
NACTIVE	INTEGER NOT NULL	Number of active pages in the partition.	S
PCTROWCOMP	SMALLINT NOT NULL	Percentage of rows compressed within the total number of active rows in the partition. This includes any row in a table space that is defined with COMPRESS YES.	S
STATSTIME	TIMESTAMP NOT NULL	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
DBNAME	VARCHAR(24) NOT NULL	Database that contains the table space named in TSNAME.	G
TSNAME	VARCHAR(24) NOT NULL	Table space that contains the table.	G
PARTITION	SMALLINT NOT NULL	Partition number of the table space that contains the table.	G
OWNER	VARCHAR(128) NOT NULL	The schema of the table.	G
NAME	VARCHAR(128) NOT NULL	Name of the table.	G
CARDF	FLOAT NOT NULL WITH DEFAULT -1	Total number of rows in the partition.	S

SYSIBM.SYSTABSTATS_HIST table

The SYSIBM.SYSTABSTATS_HIST table contains rows from SYSTABSTATS.

Rows are added or changed in this table when RUNSTATS collects history statistics. Rows in this table can also be inserted, updated, and deleted.

Column name	Data type	Description	Use
NPAGES	INTEGER NOT NULL	Total number of pages on which rows of the partition appear.	S
STATSTIME	TIMESTAMP NOT NULL	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics.	G
DBNAME	VARCHAR(24) NOT NULL	Database that contains the table space named in TSNAME.	G
TSNAME	VARCHAR(24) NOT NULL	Table space that contains the table.	G
PARTITION	SMALLINT NOT NULL	Partition number of the table space that contains the table.	G
OWNER	VARCHAR(128) NOT NULL	The schema of the table.	G
NAME	VARCHAR(128) NOT NULL	Name of the table.	G
CARDF	FLOAT(8) NOT NULL WITH DEFAULT -1	Total number of rows in the partition. The value is -1 if statistics have not been gathered.	S
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

SYSIBM.SYSTRIGGERS table

The SYSIBM.SYSTRIGGERS table contains one row for each trigger.

Column name	Data type	Description	Use
NAME	VARCHAR(128) NOT NULL	Name of the trigger and trigger package.	G
SCHEMA	VARCHAR(128) NOT NULL	Schema of the trigger. This implicit or explicit qualifier for the trigger name is also used for the collection ID of the trigger package.	G
	SMALLINT NOT NULL	Not used.	N
DBID	SMALLINT NOT NULL	Internal identifier of the database for the trigger.	G
OBID	SMALLINT NOT NULL	Internal identifier of the trigger.	G
OWNER	VARCHAR(128) NOT NULL	Owner of the trigger.	G
CREATEDBY	VARCHAR(128) NOT NULL	Primary authorization ID of the user who created the trigger.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table or view.	G
TBOWNER	VARCHAR(128) NOT NULL	Qualifier of the name of the table or view to which this trigger applies.	G
TRIGTIME	CHAR(1) NOT NULL	Time when triggered actions are applied to the base table, relative to the event that activated the trigger: **A** Trigger is applied after the event. **B** Trigger is applied before the event. **I** Trigger is applied instead of the event	G
TRIGEVENT	CHAR(1) NOT NULL	Operation that activates the trigger: **I** Insert **D** Delete **U** Update	G
GRANULARITY	CHAR(1) NOT NULL	Trigger is executed once per: **S** Statement **R** Row	G
CREATEDTS	TIMESTAMP NOT NULL	Time when the CREATE statement was executed for this trigger. The time value is used in resolving functions, distinct types, and stored procedures. It is also used to order the execution of multiple triggers.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies. RELCREATED should be used instead.	G

Column name	Data type	Description	Use
	VARCHAR(6000) NOT NULL	Not used.	N
REMARKS	VARCHAR(762) NOT NULL	A character string provided by the user with the COMMENT statement.	G
TRIGNAME	VARCHAR(128) NOT NULL	Unused	G
OWNERTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of creator: **blank** Authorization ID **L** Role	G
ENVID	INTEGER NOT NULL WITH DEFAULT	Internal environment identifier.	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. Blank if created prior to Version 9. See Release dependency indicators for all other values.	G
	CHAR(1) NOT NULL WITH DEFAULT	Reserved for IBM use.	I
	CHAR(1) NOT NULL WITH DEFAULT	Reserved for IBM use.	I
	INTEGER NOT NULL	Reserved for IBM use.	I
	VARCHAR(96) NOT NULL	Reserved for IBM use.	I
SECURE	CHAR(1) NOT NULL WITH DEFAULT 'N'	Indicates if the trigger is secured: **N** The trigger is not secured **Y** The trigger is secured	G
ALTEREDTS	TIMESTAMP NOT NULL	Time when the last ALTER statement was executed for this trigger.	G
ROWID	ROWID NULL GENERATED ALWAYS	ROWID column, created for the lob columns in this table.	G
STATEMENT	CLOB(2M) NOT NULL WITH DEFAULT	The text of the entire CREATE TRIGGER statement that was used to create the object.	G

SYSIBM.SYSTRIGGERS_STMT table

The SYSIBM.SYSTRIGGERS_STMT table is an auxiliary table for the STATEMENT column of the SYSIBM.SYSTRIGGERS table and is required to hold LOB data.

Column name	Data type	Description	Use
STATEMENT	CLOB(2M) NOT NULL WITH DEFAULT	The text of the entire CREATE TRIGGER statement that was used to create the object.	G

SYSIBM.SYSUSERAUTH table

The SYSIBM.SYSUSERAUTH table records the system privileges that are held by users.

Column name	Data type	Description	Use
GRANTOR	VARCHAR(128) NOT NULL	Authorization ID of the user who granted the privileges.	G
GRANTEE	VARCHAR(128) NOT NULL	Authorization ID of the user that holds the privilege. Could also be PUBLIC for a grant to PUBLIC.	G
	CHAR(12) NOT NULL	Internal use only	I
	CHAR(6) NOT NULL	Not used	N
	CHAR(8) NOT NULL	Not used	N
	CHAR(1) NOT NULL	Not used	N
AUTHHOWGOT	CHAR(1) NOT NULL WITH DEFAULT	Authorization level of the user from whom the privileges were received. This authorization level is not necessarily the highest authorization level of the grantor. **blank** Not applicable **C** DBCTRL **D** DBADM **E** SECADM **G** ACCESSCTRL **K** SQLADM **L** SYSCTRL **M** DBMAINT **O** SYSOPR **S** SYSADM	G
	CHAR(1) NOT NULL	Not used	N
BINDADDAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can use the BIND subcommand with the ADD option: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G
BSDSAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can issue the RECOVER BSDS command: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G

Column name	Data type	Description	Use
CREATEDBAAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can create databases and automatically receive DBADM authority over the new databases: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G
CREATEDBCAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can execute the CREATE DATABASE statement to create new databases and automatically receive DBCTRL authority over the new databases: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G
CREATESGAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can execute the CREATE STOGROUP statement to create new storage groups: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G
DISPLAYAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can use the DISPLAY commands: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G
RECOVERAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can use the RECOVER INDOUBT command: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G
STOPALLAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can use the STOP command: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G
STOSPACEAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can use the STOSPACE utility: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G
SYSADMAUTH	CHAR(1) NOT NULL	Whether the GRANTEE has system administration authority: **blank** Privilege is not held **G** Privilege was granted with the GRANT option **Y** Privilege was granted without the GRANT option GRANTEE has the privilege with the GRANT option for a value of either Y or G.	G
SYSOPRAUTH	CHAR(1) NOT NULL	Whether the GRANTEE has system operator authority: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G
TRACEAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can issue the START TRACE and STOP TRACE commands: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G

Column name	Data type	Description	Use
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
MON1AUTH	CHAR(1) NOT NULL WITH DEFAULT	Whether the GRANTEE can obtain IFC serviceability data: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G
MON2AUTH	CHAR(1) NOT NULL WITH DEFAULT	Whether the GRANTEE can obtain IFC data: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G
CREATEALIASAUTH	CHAR(1) NOT NULL WITH DEFAULT	Whether the GRANTEE can execute the CREATE ALIAS statement: **blank** Privilege is not held **G** Privilege held with the GRANT option **Y** Privilege held without the GRANT option	G
SYSCTRLAUTH	CHAR(1) NOT NULL WITH DEFAULT	Whether the GRANTEE has SYSCTRL authority: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option GRANTEE has the privilege with the GRANT option for a value of either Y or G.	G
BINDAGENTAUTH	CHAR(1) NOT NULL WITH DEFAULT	Whether the GRANTEE has BINDAGENT privilege: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G
ARCHIVEAUTH	CHAR(1) NOT NULL WITH DEFAULT	Whether the GRANTEE is privileged to use the ARCHIVE LOG command: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G
	CHAR(1) NOT NULL WITH DEFAULT	Not used	N
	CHAR(1) NOT NULL WITH DEFAULT	Not used	N
GRANTEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the GRANT statement was executed. The value is '1985-04-01.00.00.00.000000' for the one installation row.	G
CREATETMTAB-AUTH	CHAR(1) NOT NULL WITH DEFAULT	Whether the GRANTEE has CREATETMTABAUTH privilege: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G

Column name	Data type	Description	Use
GRANTEETYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of grantee: **blank** Authorization ID **L** Role	G
GRANTORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of grantor: **blank** Authorization ID **L** Role	G
DEBUGSESSION-AUTH	CHAR(1) NOT NULL WITH DEFAULT	Whether the GRANTEE has DEBUGSESSION privilege: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G
EXPLAINAUTH	CHAR(1) NOT NULL WITH DEFAULT	Whether the GRANTEE can explain and prepare statements: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G
SQLADMAUTH	CHAR(1) NOT NULL WITH DEFAULT	Whether the GRANTEE has SQLADM authority: **blank** Privilege is not held **G** Privilege is held with the GRANT option **Y** Privilege is held without the GRANT option	G
SDBADMAUTH	CHAR(1) NOT NULL WITH DEFAULT	Whether the GRANTEE has system DBADM authority: **blank** Privilege is not held **Y** Privilege is held without the GRANT option	G
DATAACCESSAUTH	CHAR(1) NOT NULL WITH DEFAULT	Whether the GRANTEE has DATAACCESS authority: **blank** Privilege is not held **Y** Privilege is held without the GRANT option	G
ACCESSCTRLAUTH	CHAR(1) NOT NULL WITH DEFAULT	Whether the GRANTEE has ACCESSCTRL authority: **blank** Privilege is not held **Y** Privilege is held without the GRANT option	G
CREATESECURE-AUTH	CHAR(1) NOT NULL WITH DEFAULT	Whether the GRANTEE can create secured objects (triggers and user-defined functions): **blank** Privilege is not held **Y** Privilege is held without the GRANT option	G

SYSIBM.SYSVARIABLES table

The SYSIBM.SYSVARIABLES table contains one row for each global variable that is created.

Column name	Data type	Description	Use
VARID	BIGINT NOT NULL GENERATED ALWAYS AS IDENTITY	The identifier of the global variable.	G
SCHEMA	VARCHAR(128) NOT NULL	The schema name of the global variable.	G
NAME	VARCHAR(128) NOT NULL	The unqualified name of the global variable.	G
OWNER	VARCHAR(128) NOT NULL	The authorization ID of the owner of the global variable.	G
OWNERTYPE	CHAR(1) NOT NULL	The type of owner of the global variable: **L** The owner is a role **blank** The owner is an authorization ID	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. See Release dependency indicators for all other values.	G
CREATEDTS	TIMESTAMP NOT NULL	Time at which the global variable was created.	G
TYPESCHEMA	VARCHAR(128) NOT NULL	The schema name of the data type. For built-in data types, this value is SYSIBM.	G
TYPENAME	VARCHAR(128) NOT NULL	The unqualified name of the data type.	G
DATATYPEID	INTEGER NOT NULL	For a built-in data type, the internal ID of the built-in type. For a distinct type, the internal ID of the distinct type.	S
SOURCETYPEID	INTEGER NOT NULL	For a built-in data type, 0. For a distinct type, the internal ID of the built-in data type on which the distinct type is based.	S
LENGTH	INTEGER NOT NULL	The maximum length of the global variable.	G
SCALE	SMALLINT NOT NULL	The scale of the global variable.	G
CCSID	INTEGER NOT NULL	The CCSID of the global variable.	G

Column name	Data type	Description	Use
DEFAULT	CHAR(3) NOT NULL	The default value of the global variable. This column can contain one of the following values: **N** The global variable does not have a default value. **S** The default value is the value of the SQL authorization ID of the process at the time that a default value is used. **1** The default value is a string constant. **2** The default value is a floating-point constant. **3** The default value is a decimal constant. **4** The default value is an integer constant. **5** The default value is a hexadecimal character string. **6** The default value is a UX string. **7** The global variable has a graphic data type and has a default value that is a character string constant. **8** The global variable has a character data type and has a default value that is a character string constant. **9** The default value is a DECFLOAT constant If this column contains one of the following values, the default value of the global variable is the value of the indicated special register at the time that a default value is used: **AES** CURRENT APPLICATION ENCODING SCHEME **ACT** CURRENT CLIENT_ACCTNG **APN** CURRENT CLIENT_APPLNAME **CID** CURRENT CLIENT_USERID **WSN** CURRENT CLIENT_WRKSTNNAME **DAT** CURRENT DATE **DBG** CURRENT DEBUG MODE **DEC** CURRENT DECFLOAT ROUNDING MODE **DEG** CURRENT DEGREE **EXP** CURRENT EXPLAIN MODE **LCT** CURRENT LOCALE LC_CTYPE **MTT** CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION **MEM** CURRENT MEMBER **HNT** CURRENT OPTIMIZATION HINT **CPP** CURRENT PACKAGE PATH **CPS** CURRENT PACKAGESET **PTH** CURRENT PATH **PRC** CURRENT PRECISION **RFA** CURRENT REFRESH AGE **RVS** CURRENT ROUTINE VERSION **RUL** CURRENT RULES **SCH** CURRENT SCHEMA **SVR** CURRENT SERVER **TIM** CURRENT TIME **TST** CURRENT TIMESTAMP **STZ** SESSION TIME ZONE **U** SESSION_USER	G

Column name	Data type	Description	Use
ROWID	ROWID NOT NULL GENERATED ALWAYS	The ROWID value for the lob columns in this table.	G
DEFAULTTEXT	CLOB(2M) NOT NULL WITH DEFAULT	The text of the default value of the global variable.	G
	BLOB(2M) NOT NULL	Reserved for IBM use.	I
ENVID	INTEGER NOT NULL	Internal environment identifier.	G
REMARKS	VARCHAR(762) NOT NULL	A character string about this global variable that is provided by using the COMMENT statement.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies. RELCREATED should be used instead.	G

SYSIBM.SYSVARIABLEAUTH table

The SYSIBM.SYSVARIABLEAUTH table contains one row for each privilege of each authorization ID that has privileges on a global variable.

Column name	Data type	Description	Use
GRANTOR	VARCHAR(128) NOT NULL	The grantor of the privilege.	G
GRANTORTYPE	CHAR(1) NOT NULL	The type of grantor: **blank** Grantor is an authorization ID **L** Grantor is a role	G
GRANTEE	VARCHAR(128) NOT NULL	The holder of the privilege.	G
GRANTEETYPE	CHAR(1) NOT NULL	The type of grantee: **blank** Grantee is an authorization ID **L** Grantee is a role **P** Grantee is a package. The grantee is a package if COLLID is a value other than blank.	G
SCHEMA	VARCHAR(128) NOT NULL	The schema name of the global variable.	G
NAME	VARCHAR(128) NOT NULL	The unqualified name of the global variable.	G
COLLID	VARCHAR(128) NOT NULL	If the grantee is a package, this value is the COLLID of the package.	G
CONTOKEN	CHAR(8) NOT NULL FOR BIT DATA	If the grantee is a package, this value is the consistency token of the DBRM from which the package is derived. Otherwise, this value is blank.	G
READAUTH	CHAR(1) NOT NULL	The privilege to read the global variable: **blank** The READ privilege is not held **G** The READ privilege is held with the GRANT option **Y** The READ privilege is held without the GRANT option	G
WRITEAUTH	CHAR(1) NOT NULL	The privilege to write to the global variable: **blank** The WRITE privilege is not held **G** The WRITE privilege is held with the GRANT option **Y** The WRITE privilege is held without the GRANT option	G

Column name	Data type	Description	Use
AUTHHOWGOT	CHAR(1) NOT NULL	The authorization level of the user who granted the privileges: **blank** Not applicable **E** SECADM **G** ACCESSCTRL **S** SYSADM **T** DATAACCESS This authorization level is not necessarily the highest authority level of the grantor.	G
GRANTEDTS	TIMESTAMP NOT NULL	The time when the GRANT statement was executed.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

SYSIBM.SYSVARIABLES_DESC table

The SYSIBM.SYSVARIABLES_DESC table is an auxiliary table for the SYSIBM.SYSVARIABLES table.

Column name	Data type	Description	Use
	BLOB(2M)	IBM-internal use only.	I

SYSIBM.SYSVARIABLES_TEXT table

The SYSIBM.SYSVARIABLES_TEXT table is an auxiliary table for the DEFAULTTEXT column of the SYSIBM.SYSVARIABLES table.

Column name	Data type	Description	Use
DEFAULTTEXT	CLOB(2M)	The text of the default value of the global variable.	G

SYSIBM.SYSVIEWDEP table

The SYSIBM.SYSVIEWDEP table records the dependencies of views on tables, functions, and other views.

Column name	Data type	Description	Use
BNAME	VARCHAR(128) NOT NULL	Name of the object on which the view is dependent. If the object type is a function (BTYPE='F'), the name is the specific name of the function.	G
BCREATOR	VARCHAR(128) NOT NULL	Authorization ID of the owner of BNAME.	G
BTYPE	CHAR(1) NOT NULL	Type of object: **F** Function **G** Created global temporary table **M** Materialized query table **T** Table **V** View **W** SYSTEM_TIME period **Z** BUSINESS_TIME period	G
DNAME	VARCHAR(128) NOT NULL	Name of the view.	G
DCREATOR	VARCHAR(128) NOT NULL	The schema of the view.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G
BSCHEMA	VARCHAR(128) NOT NULL WITH DEFAULT	Schema of BNAME.	G
DTYPE	CHAR(1) NOT NULL	Type of table: **F** SQL function **M** Materialized query table **V** View	G
DOWNER	VARCHAR(128) NOT NULL WITH DEFAULT	Authorization ID of the owner of the view, blank for views that were created in a DB2 release prior to Version 9.	G
OWNERTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of owner: **blank** Authorization ID **L** Role	G

SYSIBM.SYSVIEWS table

The SYSIBM.SYSVIEWS table contains one or more rows for each view, materialized query table, or user-defined SQL function.

Column name	Data type	Description	Use
NAME	VARCHAR(128) NOT NULL	Name of the object.	G
CREATOR	VARCHAR(128) NOT NULL	The schema of the object.	G
	SMALLINT NOT NULL	Not used	N
CHECK	CHAR(1) NOT NULL	Whether the WITH CHECK OPTION clause was specified in the CREATE VIEW statement: **N** No **C** Yes with the *cascaded* semantic **Y** Yes with the *local* semantic The value is N if the view has no WHERE clause, or the object is not a view.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies. RELCREATED should be used instead.	G
	VARCHAR(1500) NOT NULL	Not used	N
PATHSCHEMAS	VARCHAR(2048) NOT NULL WITH DEFAULT	SQL path at the time the object was defined. The path is used to resolve unqualified data type and function names used in the object definition.	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. Blank if created prior to Version 9. See Release dependency indicators for all other values.	G
TYPE	CHAR(1) NOT NULL	Type of table: **F** SQL function **M** Materialized query table **V** View	G
REFRESH	CHAR(1) NOT NULL WITH DEFAULT	Refresh mode: **D** A materialized query table with a deferred refresh mode **blank** Not a materialized query table	G
ENABLE	CHAR(1) NOT NULL WITH DEFAULT	Indicates whether query optimization is enabled: **Y** Enabled **N** Disabled **blank** Not a materialized query table	G
MAINTENANCE	CHAR(1) NOT NULL WITH DEFAULT	Maintenance mode: **S** For a REFRESH = 'D', a materialized query table that is maintained by the system. **U** For a REFRESH = 'D', a materialized query table that is maintained by the user. **blank** Not a materialized query table.	G

Column name	Data type	Description	Use
REFRESH_TIME	TIMESTAMP NOT NULL WITH DEFAULT	For REFRESH = 'D' and MAINTENANCE = 'S', the timestamp of the REFRESH TABLE statement that last refreshed the data. Otherwise, this is the default timestamp ('0001-01-01.00.00.00.000000').	G
ISOLATION	CHAR(1) NOT NULL WITH DEFAULT	Isolation level when the materialized query table is created or altered from a base table: **R** RR (repeatable read) **S** CS (cursor stability) **T** RS (read stability) **U** UR (uncommitted read) **blank** Not a materialized query table	G
SIGNATURE	VARCHAR(1024) NOT NULL WITH DEFAULT FOR BIT DATA	Contains an internal description. Used for materialized query tables.	G
APP_ENCODING_ CCSID	INTEGER NOT NULL WITH DEFAULT	CCSID of the current application encoding scheme at the time the object was created. For objects created prior to Version 8 of DB2, the value is 0.	G
OWNER	VARCHAR(128) NOT NULL WITH DEFAULT	Authorization ID of the owner of the view, blank for views that were created in a DB2 release prior to Version 9.	G
OWNERTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of owner: **blank** Authorization ID **L** Role	G
ENVID	INTEGER NOT NULL WITH DEFAULT	Internal environment identifier.	G
ROWID	ROWID NULL GENERATED ALWAYS	ROWID column, created for the lob columns in this table	G
STATEMENT	CLOB(2M) NOT NULL WITH DEFAULT	The text of the entire CREATE VIEW statement that was used to create the object.	G
	BLOB(1G) NOT NULL WITH DEFAULT	Internal use only.	I

SYSIBM.SYSVIEWS_STMT table

The SYSIBM.SYSVIEWS_STMT table is an auxiliary table for the STATEMENT column of the SYSIBM.SYSVIEWS table and is required to hold LOB data.

Column name	Data type	Description	Use
STATEMENT	CLOB(2M) NOT NULL WITH DEFAULT	The text of the statement that was used to create the object.	G

SYSIBM.SYSVIEWS_TREE table

The SYSIBM.SYSVIEWS_TREE table is an auxiliary table for the PARSETREE column of the SYSIBM.SYSVIEWS table and is required to hold LOB data.

Column name	Data type	Description	Use
	BLOB(1G) NOT NULL WITH DEFAULT	Internal use only.	I

SYSIBM.SYSVOLUMES table

The SYSIBM.SYSVOLUMES table contains one row for each volume of each storage group.

Column name	Data type	Description	Use
SGNAME	VARCHAR(128) NOT NULL	Name of the storage group.	G
SGCREATOR	VARCHAR(128) NOT NULL	Authorization ID of the owner of the storage group.	G
VOLID	VARCHAR(18) NOT NULL	Serial number of the volume or * if SMS-managed.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies. RELCREATED should be used instead.	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. Blank if created prior to Version 9. See Release dependency indicators for all other values.	G

SYSIBM.SYSXMLRELS table

The SYSIBM.SYSXMLRELS table contains one row for each XML table that is created for an XML column.

Column name	Data type	Description	Use
TBOWNER	VARCHAR(128) NOT NULL	Schema or qualifier of the base table.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the base table.	G
COLNAME	VARCHAR(128) NOT NULL	Name of the XML column in the base table.	G
XMLTBOWNER	VARCHAR(128) NOT NULL	Schema or qualifier of the XML table.	G
XMLTBNAME	VARCHAR(128) NOT NULL	Name of the XML table.	G
XMLRELOBID	INTEGER NOT NULL	Internal identifier of the relationship between the base table and the XML table.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies. RELCREATED should be used instead.	G
CREATEDTS	TIMESTAMP NOT NULL	Time when the XML table was created.	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. See Release dependency indicators for the values.	G

SYSIBM.SYSXMLSTRINGS table

Each row of the SYSIBM.SYSXMLSTRINGS table contains a single string and its unique ID that are used to condense XML data. The string can be an element name, attribute name, name space prefix, or a namespace URI.

Column name	Data type	Description	Use
STRINGID	INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY	Unique ID for the string.	G
STRING	VARCHAR(1000) NOT NULL	The string data.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

SYSIBM.USERNAMES table

Each row in the SYSIBM.USERNAMES table is used to carry out one outbound ID translation or inbound ID translation and "come from" checking.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
TYPE	CHAR(1) NOT NULL	How the row is to be used: **I** For inbound translation and "come from" checking. **O** For outbound translation. **S** For outbound system AUTHID to establish a trusted connection.	G
AUTHID	VARCHAR(128) NOT NULL WITH DEFAULT	Authorization ID to be translated. Applies to any authorization ID if blank.	G
LINKNAME	VARCHAR(24) NOT NULL	Identifies the VTAM or TCP/IP network locations associated with this row. A blank value in this column indicates this name translation rule applies to any TCP/IP or SNA partner. If a non-blank LINKNAME is specified, one or both of the following statements must be true: • A row exists in SYSIBM.LUNAMES whose LUNAME matches the value specified in the SYSIBM.USERNAMES LINKNAME column. This row specifies the VTAM site associated with this name translation rule. • A row exists in SYSIBM.IPNAMES whose LINKNAME matches the value specified in the SYSIBM.USERNAMES LINKNAME column. This row specifies the TCP/IP host associated with this name translation rule. Inbound name translation and "come from" checking are not performed for TCP/IP clients.	G
NEWAUTHID	VARCHAR(128) NOT NULL WITH DEFAULT	Translated value of AUTHID. Blank specifies no translation.NEWAUTHID can be stored as encrypted data by calling the DSNLEUSR stored procedure. To send the encrypted value of AUTHID across a network, one of the encryption security options in the SYSIBM.IPNAMES table should be specified.	G
PASSWORD	VARCHAR(255) NOT NULL WITH DEFAULT	Password to accompany an outbound request, if passwords are not encrypted by RACF. If passwords are encrypted, or the row is for inbound requests, the column is not used.PASSWORD can be stored as encrypted data by calling the DSNLEUSR stored procedure. To send the encrypted value of PASSWORD across a network, one of the encryption security options in the SYSIBM.IPNAMES table should be specified.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies.	G

SYSIBM.SYSXMLTYPMOD table

The SYSIBM.SYSXMLTYPMOD table contains rows about the XML type modifiers of XML columns. Rows in this table can be inserted, updated and deleted.

Column name	Data type	Description	Use
XML_TYPEMOD _ID	INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY	An id generated for the XML type modifier, it is an identity column and primary key.	G
TYPE_ ANNOTATION	CHAR(1) NOT NULL	Indicate whether there is type annotation. **Y** WITH type annotation **N** with no type annotation.	G
CREATEDTS	TIMESTAMP NOT NULL	The timestamp when this type modifier is created.	G
ALTEREDTS	TIMESTAMP NOT NULL	The timestamp when this type modifier is altered	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. See Release dependency indicators for the values.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies. RELCREATED should be used instead.	G
CREATEDBY	VARCHAR(128) NOT NULL	Primary authorization ID of the user who created the database.	G

SYSIBM.SYSXMLTYPMSCHEMA table

The SYSIBM.SYSXMLTYPMSCHEMA table contains the XML schema information for an XML type modifier. It contains one row per XML schema for an XML type modifier.

Column name	Data type	Description	Use
XML_TYPEMOD _ID	INTEGER NOT NULL	The id for the XML type modifier.	G
XSROBJECTID	INTEGER NOT NULL	The id for an XML schema registered in XSR.	G
ELEMENT_ NAMESPACE	INTEGER NOT NULL	String id for the namespace name of the root element node. By default, it is the TARGETNAMESPACE of the XML schema. It would be 0 if it is NO NAMESPACE.	G
ELEMENT_ NAME	INTEGER NOT NULL	String id for the local name of the root element node. It would be 0 if it is not specified.	G
CREATEDTS	TIMESTAMP NOT NULL	The timestamp when this type modifier is created.	G
ALTEREDTS	TIMESTAMP NOT NULL	The timestamp when this type modifier is altered	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. See Release dependency indicators for the values.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators. The value in this field is not a reliable indicator of release dependencies. RELCREATED should be used instead.	G

DB2 directory tables

DB2 for z/OS maintains a set of tables (in database DSNDB01) called the DB2 directory.

About these topics

These topics describe the directory tables that allow SELECT operations by describing the columns of those tables.

Authorized users can query the directory; however, it is primarily intended for use by DB2 and is therefore subject to change.

Users must have one of the following privileges to execute SELECT statements on the directory tables:
- Installation SYSADM
- SYSADM
- SYSCTRL
- ACCESSCTRL
- DATAACCESS
- SECADM
- SQLADM
- System DBADM
- DBADM on DSNDB01
- The SELECT privilege on a specific table

All directory tables are qualified by SYSIBM. Do not use this qualifier for user-defined tables.

The directory tables are updated by DB2 during normal operations in response to certain SQL statements, commands, and utilities.

Programming interface information

Not all directory table columns are part of the general-use programming interface. Whether a column is part of this interface is indicated in a column labeled "Use" in the table that describes the column. The values that "Use" can assume are as follows:

Value	Meaning
S	Column is part of the product-sensitive interface
I	Column is for internal use only

For columns for which "Use" is I, the name of the column and its description do not appear in the explanation of the column.

Directory table spaces and indexes

DB2 directory tables are contained in certain table spaces and have indexes.

The following table lists the table space and indexes for each directory table and lists the index fields for each index. The indexes are in ascending order.

The directory table space, tables, and indexes are primarily intended for use by DB2 and are therefore subject to change.

Table 177. Table spaces and indexes for the directory tables

TABLE SPACE DSNDB01. ...	TABLE SYSIBM. ...	INDEX SYSIBM. ...	INDEX FIELDS
DBD01	DBDR	DSNDB01X	DBID.SECTION
SYSDBDXA	SYSDBD_DATA	DSNDB1XA	DBD_DATA
SCT02	SCTR	DSNSCT02	SCTNAME.SCTSEC. SPTSEQ
SPT01	SPTR	DSNSPT01	SPTPID.SPTSEC. SPTSEQ
		DSNSPT02	version.SPTID. SPTSEC.SPTSEQ
SYSSPUXA	SYSSPTSEC_DATA	DSNSPDXA	SPTSEC_DATA
SYSSPUXB	SYSSPTSEC_EXPL	DSNSPEXA	SPTSEC_EXPL
SYSLGRNX	SYSLGRNX	DSNLLX01	LGRDBID.LGRPSID. LGRPART.LGRMEMB. LGRSLRSN
		DSNLLX02	LGRDBID.LGRPSID. LGRSLRSN
SYSUTILX	SYSUTIL	DSNLUX01	USUUID
	SYSUTILX	DSNLUX02	UTILID.SEQNO

SYSIBM.DBDR table

The DBDR table stores one row for each DBD section.

Column name	Data type	Description	Use
	INTEGER	Not used	S
DBID	SMALLINT	DBID of the database	S
SECTION	SMALLINT	DBD section number	S
DBD_ROWID	ROWID	ID that is used to support the DBD_DATA column	S
DBD_DATA	BLOB(2G)	DBD data for the section	I

SYSIBM.SYSDBD_DATA table

The SYSIBM.SYSDBD_DATA table is an auxiliary table for the SYSIBM.DBDR table.

Column name	Data type	Description	Use
DBD_DATA	BLOB(2G)	Contents of the DBD section.	I

SYSIBM.SCTR table

The SYSIBM.SCTR table stores Skeleton Cursor Tables (SKCT) information.

Column name	Data type	Description	Use
SCTLL	CHAR (4) FOR BIT DATA	The length of the record.	S
SCTNAME	CHAR (14) FOR BIT DATA	The plan name, section number, and sequence number.	S
SCTDAT	VARCHAR(4028)	SKCT data.	I

SYSIBM.SPTR table

The SYSIBM.SPTR table stores Skeleton Package Table (SKPT) information

Column name	Data type	Description	Use
SPTLL	INTEGER		S
SPTLOCID	VARCHAR(128)		S
SPTCOLID	VARCHAR(128)		S
SPTNAME	VARCHAR(128)		S
SPTCONID	CHAR(8) FOR BIT DATA		S
SPTRESV	CHAR(2) FOR BIT DATA		S
SPTSEC	CHAR(4) FOR BIT DATA		S
SPTSEQ	CHAR(2) FOR BIT DATA		S
SPTBODY	VARCHAR(1)		S
SPTVER	VARCHAR(64)		S
SPT_ROWID	ROWID		S
SPT_DATA	BLOB(2G)		I
SPT_EXPLAIN	BLOB(2G)		I

SYSIBM.SYSSPTSEC_DATA table

The SYSIBM.SYSSPTSEC_DATA table is an auxiliary table that contains package data for the SYSIBM.SPTR table.

Column name	Data type	Description	Use
SPT_DATA	BLOB(2G)	Contents of the SKPT section	I

SYSIBM.SYSSPTSEC_EXPL table

The SYSIBM.SYSSPTSEC_EXPL table is an auxiliary table that contains static package explain data for the SYSIBM.SPTR table.

Column name	Data type	Description	Use
SPT_EXPLAIN	BLOB(2G)	Contents of the SKPT section explain block	I

SYSIBM.SYSLGRNX table

The SYSLGRNX table stores recovery log ranges that record the time an index space defined with COPY YES or a table space was open for updates. This provides an efficient way for DB2 to access the appropriate log records for recovery, rather than having to scan every record in the recovery log for a particular table.

Column name	Data type	Description	Use
LGRDBID	CHAR(2) FOR BIT DATA	DBID of the modified object	S
LGRPSID	CHAR(2) FOR BIT DATA	OBID of the modified object	S
LGRUCDT	CHAR(6)	Modification date in the form *mmddyy*	S
LGRUCTM	CHAR(8)	Modification time in the form *hhmmssth*	S
LGRSRBA[1]	CHAR(10) FOR BIT DATA	Starting RBA	S
LGRSPBA[1]	CHAR(10) FOR BIT DATA	Stopping RBA	S
LGRPART	SMALLINT	Partition number in the table space or index space	S
LGRSLRSN[1]	CHAR(10) FOR BIT DATA	Starting LRSN of update log records for data sharing. Otherwise, the system clock value that corresponds to the first update log record.	S
LGRELRSN[1]	CHAR(10) FOR BIT DATA	Ending LRSN of update log records for data sharing. Otherwise, the system clock value that corresponds to the last update log record.	S
LGRMEMB	CHAR(2)	Data sharing member ID of the modifying DB2 subsystem. X'0000' for a non-data-sharing environment.	S

Note:

1. A SELECT from SYSIBM.SYSLGRNX displays this column in either 6-byte or 10-byte format. Before CATENFM of SYSLGRNX the data and the display are in 6-byte format. After CATENFM of SYSLGRNX the data and the display are in 10-byte format.

SYSIBM.SYSUTIL table

The SYSUTIL table stores status information about DB2 utilities that are active or stopped. Each record is uniquely identified by the utility identifier. Each row of the table contains the information for one utility execution step. When the utility completes, the corresponding entries in the SYSUTIL table are deleted.

Name	Data type	Description	Use
USUUID	CHAR(16)	UTILID value that was passed in a JOB statement parameter	S
USUJOBNM	CHAR(8)	Job name from the JOB statement	S
USUAUID	CHAR(8)	Authorization ID of the invoker	S
USURDATE	CHAR(4) FOR BIT DATA	Date of the utility	S
USUREL	CHAR(3)	Utility release level at restart time	S
USUIRQD	CHAR(1)	IBM required field	S
USULSIZE	CHAR(4) FOR BIT DATA	List size	S
USULCUR	CHAR(4) FOR BIT DATA	The object that is currently being processed or was last processed	S
USUUTNAM	CHAR(8)	Name of the currently executing utility	S
USUPHASE	CHAR(8)	Current phase of the currently executing utility	S
USUDSNU	CHAR(2) FOR BIT DATA	Data set or piece number	S
USUDSNU2	CHAR(2) FOR BIT DATA	Ending number of the partition range	S
USUSTATU	CHAR(1)	Status of the currently executing utility	S
USUTREQ	CHAR(1)	Termination requested (Y or N)	S
USUFORCE	CHAR(1)	Element of USO forced (Y or N)	S
USURLOK	CHAR(1)	Reload was successful (Y or N)	S
USUCMPOK	CHAR(1)	Compatibility check passed (Y or N)	S
USURSFLG	BIT(8)	Utility restriction flags	S
USURTFLG	BIT(8)	Term settings	S
USURSFLG2	BIT(8)	Utility flags	S
USUPOS	CHAR(4) FOR BIT DATA	Relative USM position in the SYSIN DD statement	S
USUDONE	CHAR(8) FOR BIT DATA	Number of objects processed	S
USUCKSUM	CHAR(4) FOR BIT DATA	USU checksum	S
USUDBOB	CHAR(2) FOR BIT DATA	DBID for the table space	S
USUPSID	CHAR(2) FOR BIT DATA	PSID for the table space or index space	S
USUPSDD	CHAR(2) FOR BIT DATA	Secondary PSID for RECOVER INDEX data page set	S
USUCATMGFRM	CHAR(1) FOR BIT DATA	Saved catalog level for the release from which migration is done, from the DBD01 header page	S

Name	Data type	Description	Use
USUOFLAG	CHAR(1) FOR BIT DATA	Flags for object properties	S
USUDBNAM	CHAR(8)	Database name	S
USUSPNAM	CHAR(8)	Table space or index space name	S
USUMEMBR	CHAR(8)	Member name	S
USUOCATR	CHAR(1) FOR BIT DATA	Saved catalog release level, from the DBD01 header page	S
USUOCATV	CHAR(1) FOR BIT DATA	Saved catalog version level, from the DBD01 header page	S
USUOCATCV	CHAR(1) FOR BIT DATA	Saved migration mode, from the DBD01 header page	S
USUOCATH	CHAR(1) FOR BIT DATA	Saved highest version of the catalog	S
USUUDA	CHAR(150) FOR BIT DATA	Utility-dependent data	S
USURTIME	CHAR(4) FOR BIT DATA	Latest utility start time	S
USURLSN	CHAR(6) FOR BIT DATA	Latest utility start LRSN	S
USURDATO	CHAR(4) FOR BIT DATA	Original utility start date	S
USURTIMO	CHAR(4) FOR BIT DATA	Original utility start time	S
USURLSNO	CHAR(4) FOR BIT DATA	Original utility start LRSN	S
USUR5	CHAR(10) FOR BIT DATA	Reserved	I
USURCNTR	CHAR(31) FOR BIT DATA	Generic counter or value holder	S
USURLSNX	CHAR(10) FOR BIT DATA	Latest utility start LRSN value	S
USURLSOX	CHAR(10) FOR BIT DATA	Original utility start LRSN value	S
USUR6	CHAR(72) FOR BIT DATA	Reserved	I
USUUSTRN	CHAR(27000) FOR BIT DATA	Utility-dependent restart information	S

SYSIBM.SYSUTILX table

The SYSUTILX table is a dependent of the SYSUTIL table. A record is created in the SYSUTILX table when the amount of information in the parent record exceeds the record size of SYSUTIL. The rows in SYSUTILX are uniquely identified by the utility identifier and sequence number.

Column name	Data type	Description	Use
UTILID	CHAR(16)	The utility ID that identifies the parent record in SYSIBM.SYSUTIL	S
SEQNO	SMALLINT	The sequence number of this row	S
	CHAR(12)	Reserved	I
CHECKPOINT	VARCHAR(32000)	The overflow checkpoint/restart information	S

Performance information for SQL application programming

Efficient applications are an important first step to good system and application performance. As you code applications that access data in DB2, consider performance objectives in your application design.

The following topics can help you understand how application programmers can consider performance as they write applications that access data in DB2 for z/OS.

Concurrency and programming

The goal is to program and prepare applications in a way that:

- Protects the integrity of the data that is being read or updated from being changed by other applications.
- Minimizes the length of time that other access to the data is prevented.

For more information about DB2 concurrency and recommendations for improving concurrency in your application programs, see the following topics:

- Concurrency recommendations for application designers (Introduction to DB2 for z/OS)
- Concurrency and locks (DB2 Performance)
- Improving concurrency (DB2 Performance)
- Improving concurrency in data sharing environments (DB2 Data Sharing Planning and Administration)

Writing efficient queries

The predicates, subqueries, and other structures in SQL statements affect the access paths that DB2 uses to access the data.

For information about how to write SQL statements that access data efficiently, see the following topics:

- Ways to improve query performance (Introduction to DB2 for z/OS)
- Writing efficient SQL queries (DB2 Performance)

Analyzing access paths

By analyzing the access path that DB2 uses to access the data for an SQL statement, you can discover potential problems. You can use this information to modify your statement to perform better.

For information about how you can use EXPLAIN tables, and SQL optimization tools, to analyze the access paths for your SQL statements, see the following topics:

- Investigating access path problems (DB2 Performance)
- Using EXPLAIN to understand the access path (Introduction to DB2 for z/OS)
- Investigating SQL performance by using EXPLAIN (DB2 Performance)
- Interpreting data access by using EXPLAIN (DB2 Performance)
- EXPLAIN tables (DB2 Performance)
- "EXPLAIN" on page 1642
- Tuning SQL with Optim Query Tuner, Part 1: Understanding access paths
- Generating visual representations of access plans

Distributed data access performance

The goal is to reduce the amount of network traffic that is required to access the distributed data, and to manage the use of system resources such as distributed database access threads and connections.

For information about improving the performance of applications that access distributed data, see the following topics:

- Ways to reduce network traffic (Introduction to DB2 for z/OS)
- Managing DB2 threads (DB2 Performance)
- Improving performance for applications that access distributed data (DB2 Performance)
- Improving performance for SQL statements in distributed applications (DB2 Performance)

Stored procedures performance

For information about stored procedures and DB2 performance, see the following topics:

- Implementing DB2 stored procedures (DB2 Administration Guide)
- Improving the performance of stored procedures and user-defined functions (DB2 Performance)

Related concepts:

Structured query language (Introduction to DB2 for z/OS)

Application programming for DB2 (Introduction to DB2 for z/OS)

Related tasks:

Programming applications for performance (DB2 Performance)

Planning for and designing DB2 applications (DB2 Application programming and SQL)

DB2 XML schema repository tables

The DB2 for z/OS XML schema repository (XSR) is a set of DB2 tables where you can store XML schemas.

DB2 creates the XSR tables during installation or migration. After you add XML schemas to the DB2 XSR, you can use them to validate XML documents before you store them in XML columns.

An XML schema consists of a set of XML schema documents. To add an XML schema to the DB2 XSR, you register XML schema documents to DB2. The XML schema documents must be in the Unicode encoding scheme.

Programming interface information

Not all XSR table columns are part of the general-use programming interface. Whether a column is part of this interface is indicated in a column labeled "Use" in the row that describes the table column. The meaning of the values for the "Use" column is indicated in the following table.

Table 178. Meaning of values in the "Use" column if table descriptions

Value	Meaning
G	Column is part of the general-use programming interface
S	Column is part of the product-sensitive interface
I	Column is for IBM use only
N	Column is not used

For columns for which "Use" is N or I, the name of the column and its description do not appear in the explanation of the column.

Related concepts:

XML schema management with the XML schema repository (XSR) (DB2 Programming for XML)

Related tasks:

Additional steps for enabling the stored procedures and objects for XML schema support (DB2 Installation and Migration)

Related information:

DB2-supplied stored procedures for XML schema registration and removal (DB2 Programming for XML)

XML schema repository (XSR) table spaces and indexes

DB2 XSR tables are contained in certain table spaces and have indexes.

The following tables list the table space and indexes for each XRS table and lists the index fields for each index. The indexes are in ascending order, except where noted.

Table 179. Table spaces and indexes for the DSNXSR database tables

Table space DSNXSR. ...	Table SYSIBM. ...	Index SYSIBM. ...	Index fields
SYSXSR	XSROBJECTS	XSROBJ01	XSROBJECTID
		XSROBJ02	XSROBJECTSCHEMA.XSROBJECTNAME
		XSROBJ03	TARGETNAMESPACE.SCHEMALOCATION
		XSROBJ04	SCHEMALOCATION
	XSROBJECT-COMPONENTS	XSRCOMP01	XSRCOMPONENTID
		XSRCOMP02	TARGETNAMESPACE.SCHEMALOCATION
	XSROBJECT-HIERARCHIES	XSRHIER01	XSROBJECTID.TARGETNAMESPACE. SCHEMALOCATION
		XSRHIER02	XSROBJECTID.TARGETNAMESPACE
SYSXSRA1	XSROBJECTGRAMMAR	XSRXOG01	GRAMMAR
SYSXSRA2	XSROBJECTPROPERTY	XSRXOP01	PROPERTIES
SYSXSRA3	XSRCOMPONENT	XSRXCC01	COMPONENT
SYSXSRA4	XSRPROPERTY	XSRXCP01	PROPERTIES

Note: Index field is in descending order

SYSIBM.XSRCOMPONENT table

The SYSIBM.XSRCOMPONENT table is an auxiliary table for the BLOB column COMPONENT in SYSIBM.SYSXSROBJECTCOMPONENTS. It is in LOB table space SYSXSRA3.

Column name	Data type	Description	Use
COMPONENT	BLOB(30M)	Contents of the XML schema document	G

SYSIBM.XSROBJECTS table

The SYSIBM.XSROBJECTS table contains one row for each registered XML schema.

Rows in this table can only be changed using static SQL statements issued by the DB2-supplied XSR stored procedures.

Column name	Data type	Description	Use
XSROBJECTID	INTEGER NOT NULL	Internal identifier of the XML schema. XSROBJECTID is generated as an identity column.	G
XSROBJECTSCHEMA	VARCHAR(128) NOT NULL	Qualifier of the XML schema name. This is always set to 'SYSXSR'.	G
XSROBJECTNAME	VARCHAR(128) NOT NULL	Name of the XML schema.	G
TARGETNAMESPACE	INTEGER	The value of the STRINGID column in SYSIBM.SYSXMLSTRINGS when the target namespace URI of the primary XML schema document is stored in SYSIBM.SYSXMLSTRINGS	G
SCHEMALOCATION	INTEGER	The value of the STRINGID column in SYSIBM.SYSXMLSTRINGS when the schema location URI of the primary XML schema document is stored in SYSIBM.SYSXMLSTRINGS	G
ROWID	ROWID NOT NULL GENERATED ALWAYS	The ID that is used to support BLOB data type values.	G
GRAMMAR	BLOB(250M)	The internal binary representation of the XML schema.	G
PROPERTIES	BLOB(5M)	Additional property information of the entire XML schema.	G
CREATEDBY	VARCHAR(128) NOT NULL	Authorization ID under which the XML schema was created.	G
CREATEDTS	TIMESTAMP NOT NULL	The time that the DB2-supplied stored procedure XSR_REGISTER was executed for the XML schema.	G
STATUS	CHAR(1) NOT NULL WITH DEFAULT	Registration status of the XML schema: **C** Complete **I** Incomplete **T** Temporary	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. See Release dependency indicators for the values.	G
	CHAR(1)	Not used.	N
	VARCHAR(128)	Not used.	N

Column name	Data type	Description	Use
REMARKS	VARCHAR(762)	Character string that contains comments about this XML schema.	G

SYSIBM.XSROBJECTCOMPONENTS table

The SYSIBM.XSROBJECTCOMPONENTS table contains one row for each component (document) in an XML schema.

Rows in this table can only be changed using static SQL statements issued by the DB2-supplied XSR stored procedures.

Column name	Data type	Description	Use
XSRCOMPONENTID	INTEGER NOT NULL	Internal identifier of the XML schema document. XSRCOMPONENTID is generated as an identity column.	G
TARGETNAMESPACE	INTEGER	The value of the STRINGID column in SYSIBM.SYSXMLSTRINGS when the target namespace URI of the primary XML schema document is stored in SYSIBM.SYSXMLSTRINGS.	G
SCHEMALOCATION	INTEGER	The value of the STRINGID column in SYSIBM.SYSXMLSTRINGS when the schema location URI of the primary XML schema document is stored in SYSIBM.SYSXMLSTRINGS.	G
ROWID	ROWID NOT NULL GENERATED ALWAYS	The ID that is used to support BLOB data type values.	G
COMPONENT	BLOB(30M) NOT NULL	Contents of the XML schema document.	G
PROPERTIES	BLOB(5M)	If available, additional property information of the XML schema document	G
CREATEDTS	TIMESTAMP NOT NULL	The time that the XML schema document was registered.	G
STATUS	CHAR(1) NOT NULL WITH DEFAULT	Registration status of the XML schema: **C** Complete **I** Incomplete	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. See Release dependency indicators for the values.	G

SYSIBM.XSROBJECTGRAMMAR table

SYSIBM.XSROBJECTGRAMMAR is an auxiliary table for the BLOB column GRAMMAR in SYSIBM.SYSXSROBJECTS. It is in LOB table space SYSXSRA1.

Column name	Data type	Description	Use
GRAMMAR	BLOB(250M)	Internal binary representation of the XML schema	G

SYSIBM.XSROBJECTHIERARCHIES table

The SYSIBM.XSROBJECTHIERARCHIES table contains one row for each component (document) in an XML schema to record the XML schema document hierarchy relationship.

Rows in this table can only be changed using static SQL statements issued by the DB2-supplied XSR stored procedures.

Column name	Data type	Description	Use
XSROBJECTID	INTEGER	Internal identifier of the XML schema.	G
XSRCOMPONENTID	INTEGER	Internal identifier of the XML schema document.	G
HTYPE	CHAR(1)	Hierarchy type: **D** Document **P** Primary document	G
TARGETNAMESPACE	INTEGER	The value of the STRINGID column in SYSIBM.SYSXMLSTRINGS when the target namespace URI of the primary XML schema document is stored in SYSIBM.SYSXMLSTRINGS.	G
SCHEMALOCATION	INTEGER	The value of the STRINGID column in SYSIBM.SYSXMLSTRINGS when the schema location URI of the primary XML schema document is stored in SYSIBM.SYSXMLSTRINGS.	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. See Release dependency indicators for the values.	G

SYSIBM.XSROBJECTPROPERTY table

SYSIBM.XSROBJECTPROPERTY is an auxiliary table for the BLOB column PROPERTIES in SYSIBM.SYSXSROBJECTS. It is in LOB table space SYSXSRA2.

Column name	Data type	Description	Use
PROPERTIES	BLOB(5M)	Contents of the additional property information of the entire XML schema.	G

SYSIBM.XSRPROPERTY table

The SYSIBM.XSRPROPERTY table is an auxiliary table for the BLOB column COMPONENT in SYSIBM.SYSXSROBJECTCOMPONENTS. It is in LOB table space SYSXSRA3.

Column name	Data type	Description	Use
COMPONENT	BLOB(5M)	Contents of the additional property information of the XML schema document.	G

EXPLAIN tables

EXPLAIN tables contain information about SQL statements and functions that run on DB2 for z/OS.

You can create and maintain a set of EXPLAIN tables to capture and analyze information about the performance of SQL statements and functions that run on DB2 for z/OS. Each row in an EXPLAIN table describes some aspect of a step in the execution of a query or subquery in an explainable statement. The column values for the row identify, among other things, the query or subquery, the tables and other objects involved, the methods used to carry out each step, and cost information about those methods. DB2 creates EXPLAIN output and populates EXPLAIN tables in the following situations:

- When an EXPLAIN statement is executed.
- At BIND or REBIND with the EXPLAIN(YES) or (ONLY) bind options. Rows are added for every explainable statement in the plan or package being bound. For a plan, these do not include statements in the packages that can be used with the plan. For either a package or plan, they do not include explainable statements within EXPLAIN statements nor do they include explainable statements that refer to declared temporary tables, which are incrementally bound at run time.
- When an explainable dynamic statement is executed and the value of the CURRENT EXPLAIN MODE special register is set to YES or EXPLAIN.

Related reference:

BIND and REBIND options (DB2 Commands)

Interpreting data access by using EXPLAIN (DB2 Performance)

"EXPLAIN" on page 1642

Capturing EXPLAIN information (DB2 Performance)

Creating EXPLAIN tables (DB2 Performance)

PLAN_TABLE

The plan table, PLAN_TABLE, contains information about access paths that is collected from the results of EXPLAIN statements.

Recommendation: Do not manually insert data into system-maintained EXPLAIN tables, and use care when deleting obsolete EXPLAIN table data. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools. Certain optimization tools depend on instances of the various EXPLAIN tables. Be careful not to delete data from or drop instances EXPLAIN tables that are created for these tools.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM
One instance of each EXPLAIN table can be created with the SYSIBM qualifier. DB2 and SQL optimization tools might use these tables. These tables are created when you run job DSNTIJSG when you install or migrate DB2.

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind. They are also populated when you specify EXPLAIN(YES) or EXPLAIN(ONLY) in a BIND or REBIND command. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Optional PLAN_TABLE formats

A PLAN_TABLE instance can have a format with fewer columns than those shown in the sample CREATE TABLE statement. However instances of PLAN_TABLE must have one of the following formats:

Version 11 format
All columns shown in the sample CREATE TABLE statement, up to and including the EXPANSION_REASON column (COLCOUNT=66).

Version 10 format
All columns shown in the sample CREATE TABLE statement, up to and including the MERGN column (COLCOUNT=64).

Version 9 format
All columns shown in the sample CREATE TABLE statement, to and including the PARENT_PLANNO column (COLCOUNT=59).

Version 8 format
All columns shown in the sample CREATE TABLE statement, up to and including the STMTTOKEN column (COLCOUNT=58).

Important: If the EXPLAIN tables have any format older than the Version 8 format, or are encoded in EBCDIC, DB2 returns an error for any operation that inserts rows in the EXPLAIN tables.

Column descriptions

Your subsystem or data sharing group can contain more than one of these tables, including a table with the qualifier SYSIBM, a table with the qualifier DB2OSCA, and additional tables that are qualified by user IDs.

The following table shows the descriptions of the columns in PLAN_TABLE.

Table 180. Descriptions of columns in PLAN_TABLE

Column name	Data Type	Description
QUERYNO	INTEGER NOT NULL	A number that identifies the statement that is being explained. The origin of the value depends on the context of the row: **For rows produced by EXPLAIN statements** The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax. **For rows not produced by EXPLAIN statements** DB2 assigns a number that is based on the line number of the SQL statement in the source program. When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in certain rare cases, the value is not guaranteed to be unique. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, if the QUERYNO clause is specified, then its value is used by DB2. Otherwise DB2 assigns a number based on the line number of the SQL statement in the non-inline SQL function, native SQL procedure.
QBLOCKNO	SMALLINT NOT NULL	A number that identifies each query block within a query. The value of the numbers are not in any particular order, nor are they necessarily consecutive.
APPLNAME	VARCHAR(24) NOT NULL	The name of the application plan for the row. Applies only to embedded EXPLAIN statements that are executed from a plan or to statements that are explained when binding a plan. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.

Table 180. Descriptions of columns in PLAN_TABLE (continued)

Column name	Data Type	Description
PROGNAME	VARCHAR(128) NOT NULL	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
PLANNO	SMALLINT NOT NULL	The number of the step in which the query that is indicated in QBLOCKNO was processed. This column indicates the order in which the steps were executed.
METHOD	SMALLINT NOT NULL	A number that indicates the join method that is used for the step: **0** The table in this step is the first table that is accessed, a continuation of a previous table that was accessed, or a table that is not used. **1** A nested loop join is used. For each row of the current composite table, matching rows of a new table are found and joined. **2** A merge scan join is used. The current composite table and the new table are scanned in the order of the join columns, and matching rows are joined. **3** Sorts are needed by ORDER BY, GROUP BY, SELECT DISTINCT, UNION, INTERSECT, EXCEPT, a quantified predicate, or an IN predicate. This step does not access a new table. **4** A hybrid join was used. The current composite table is scanned in the order of the join-column rows of the new table. The new table is accessed using list prefetch.
CREATOR	VARCHAR(128) NOT NULL	The creator of the new table that is accessed in this step, blank if METHOD is 3.

Table 180. Descriptions of columns in PLAN_TABLE (continued)

Column name	Data Type	Description
TNAME	VARCHAR(128) NOT NULL	The name of one of the following objects: • Table • Materialized query table • Created or declared temporary table • Materialized view • materialized table expression The value is blank if METHOD is 3. The column can also contain the name of a table in the form DSNWFQB(*qblockno*). DSNWFQB(*qblockno*) is used to represent the intermediate result of a UNION ALL, INTERSECT ALL, EXCEPT ALL, or an outer join that is materialized. If a view is merged, the name of the view does not appear. DSN_DIM_TBLX(*qblockno*) is used to the represent the work file of a star join dimension table.
TABNO	SMALLINT NOT NULL	Values are for IBM use only.
ACCESSTYPE[1]	CHAR(2) NOT NULL	The method of accessing the new table.[4]
MATCHCOLS	SMALLINT NOT NULL	For ACCESSTYPE I, IN, I1, N, NR, MX, or DX, the number of index keys that are used in an index scan; otherwise, 0.
ACCESSCREATOR	VARCHAR(128) NOT NULL	For ACCESSTYPE I, I1, N, NR, MX, or DX, the creator of the index; otherwise, blank.
ACCESSNAME	VARCHAR(128) NOT NULL	For ACCESSTYPE I, I1, H, MH, N, NR, MX, or DX, the name of the index; for ACCESSTYPE P, DSNPJW(*mixopseqno*) is the starting pair-wise join leg in MIXOPSEQ; otherwise, blank.
INDEXONLY	CHAR(1) NOT NULL	Indication of whether access to an index alone is enough to perform the step, or Indication of whether data too must be accessed. **Y** Yes **N** No
SORTN_UNIQ	CHAR(1) NOT NULL	Indication of whether the new table is sorted to remove duplicate rows. **Y** Yes **N** No
SORTN_JOIN	CHAR(1) NOT NULL	Indication of whether the new table is sorted for join method 2 or 4. **Y** Yes **N** No
SORTN_ORDERBY	CHAR(1) NOT NULL	Indication of whether the new table is sorted for ORDER BY. **Y** Yes **N** No
SORTN_GROUPBY	CHAR(1) NOT NULL	Indication of whether the new table is sorted for GROUP BY. **Y** Yes **N** No

Table 180. Descriptions of columns in PLAN_TABLE (continued)

Column name	Data Type	Description
SORTC_UNIQ	CHAR(1) NOT NULL	Indication of whether the composite table is sorted to remove duplicate rows. **Y** Yes **N** No
SORTC_JOIN	CHAR(1) NOT NULL	Indication of whether the composite table is sorted for join method 1, 2 or 4. **Y** Yes **N** No
SORTC_ORDERBY	CHAR(1) NOT NULL	Indication of whether the composite table is sorted for an ORDER BY clause or a quantified predicate. **Y** Yes **N** No
SORTC_GROUPBY	CHAR(1) NOT NULL	Indication of whether the composite table is sorted for a GROUP BY clause. **Y** Yes **N** No
TSLOCKMOD	CHAR(3) NOT NULL	An indication of the mode of lock that is acquired on either the new table, or its table space or table space partitions. If the isolation can be determined at bind time, the values are: **IS** Intent share lock **IX** Intent exclusive lock **S** Share lock **U** Update lock **X** Exclusive lock **SIX** Share with intent exclusive lock **N** UR isolation; no lock If the isolation level cannot be determined at bind time, the lock mode is determined by the isolation level at run time is shown by the following values. **NS** For UR isolation, no lock; for CS, RS, or RR, an S lock. **NIS** For UR isolation, no lock; for CS, RS, or RR, an IS lock. **NSS** For UR isolation, no lock; for CS or RS, an IS lock; for RR, an S lock. **SS** For UR, CS, or RS isolation, an IS lock; for RR, an S lock. The data in this column is right justified. For example, IX appears as a blank, followed by I, followed by X. If the column contains a blank, then no lock is acquired. If the access method in the ACCESSTYPE column is DX, DI, or DU, no latches are acquired on the XML index page and no lock is acquired on the new base table data page or row, nor on the XML table and the corresponding table spaces. The value of TSLOCKMODE is a blank in this case.

Table 180. Descriptions of columns in PLAN_TABLE (continued)

Column name	Data Type	Description
TIMESTAMP	CHAR(16) NOT NULL	This column is deprecated. Use EXPLAIN_TIME instead.
REMARKS	VARCHAR(762) NOT NULL	A field into which you can insert any character string of 762 or fewer characters. DB2 inserts a value into this column in certain situations. [6, 7]
PREFETCH	CHAR(1) NOT NULL WITH DEFAULT	Indication of whether data pages are to be read in advance by prefetch: **D** Optimizer expects dynamic prefetch **S** Pure sequential prefetch **L** Prefetch through a page list **U** List prefetch with an unsorted RID list **blank** Unknown or no prefetch
COLUMN_FN_EVAL	CHAR(1) NOT NULL WITH DEFAULT	When an SQL aggregate function is evaluated: **R** While the data is being read from the table or index **S** While performing a sort to satisfy a GROUP BY clause **blank** After data retrieval and after any sorts
MIXOPSEQ	SMALLINT NOT NULL WITH DEFAULT	The sequence number of a step in a multiple index operation. **1, 2, ...** *n* For the steps of the multiple index procedure (ACCESSTYPE is MX, MI, MU, DX, DI, or DU), the sequence number of the OR predicate in the SQL statement. (ACCESSTYPE is 'NR'). **0** For any other rows.
VERSION	VARCHAR(122) NOT NULL WITH DEFAULT	The version identifier for the package. Applies only to an embedded EXPLAIN statement executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.

Table 180. Descriptions of columns in PLAN_TABLE (continued)

Column name	Data Type	Description
COLLID	VARCHAR(128) NOT NULL WITH DEFAULT	The collection ID: **DSNDYNAMICSQLCACHE** The row originates from the dynamic statement cache **DSNEXPLAINMODEYES** The row originates from an application that specifies YES for the value of the CURRENT EXPLAIN MODE special register. **DSNEXPLAINMODEEXPLAIN** The row originates from an application that specifies EXPLAIN for the value of the CURRENT EXPLAIN MODE special register. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
ACCESS_DEGREE	SMALLINT	The number of parallel tasks or operations that are activated by a query. This value is determined at bind time; the actual number of parallel operations that are used at execution time could be different. This column contains 0 if a host variable is used. This column contains the null value if the plan or package was bound using a plan table with fewer than 43 columns. Otherwise, it can contain null if the method that it refers to does not apply.
ACCESS_PGROUP_ID[2]	SMALLINT	The identifier of the parallel group for accessing the new table. A parallel group is a set of consecutive operations, executed in parallel, that have the same number of parallel tasks. This value is determined at bind time; it could change at execution time.This column contains the null value if the plan or package was bound using a plan table with fewer than 43 columns. Otherwise, it can contain null if the method that it refers to does not apply.
JOIN_DEGREE	SMALLINT	The number of parallel operations or tasks that are used in joining the composite table with the new table. This value is determined at bind time and can be 0 if a host variable is used. The actual number of parallel operations or tasks used at execution time could be different. This column contains the null value if the plan or package was bound using a plan table with fewer than 43 columns. Otherwise, it can contain null if the method that it refers to does not apply.

Table 180. Descriptions of columns in PLAN_TABLE (continued)

Column name	Data Type	Description
JOIN_PGROUP_ID[2]	SMALLINT	The identifier of the parallel group for joining the composite table with the new table. This value is determined at bind time; it could change at execution time. This column contains the null value if the plan or package was bound using a plan table with fewer than 43 columns. Otherwise, it can contain null if the method that it refers to does not apply.
SORTC_PGROUP_ID[3]	SMALLINT	The parallel group identifier for the parallel sort of the composite table. This column contains the null value if the plan or package was bound using a plan table with fewer than 43 columns. Otherwise, it can contain null if the method that it refers to does not apply.
SORTN_PGROUP_ID[3]	SMALLINT	The parallel group identifier for the parallel sort of the new table. This column contains the null value if the plan or package was bound using a plan table with fewer than 43 columns. Otherwise, it can contain null if the method that it refers to does not apply.
PARALLELISM_ MODE[2]	CHAR(1)	The kind of parallelism, if any, that is used at bind time: **C** Query CP parallelism. **I** Query I/O parallelism. This column contains the null value if the plan or package was bound using a plan table with fewer than 43 columns, if the method that it refers to does not apply, or if the plan or package was bound prior to Version 10.
MERGE_ JOIN_ COLS	SMALLINT	The number of columns that are joined during a merge scan join (Method=2). This column contains the null value if the plan or package was bound using a plan table with fewer than 43 columns. Otherwise, it can contain null if the method that it refers to does not apply.
CORRELATION_ NAME	VARCHAR(128)	The correlation name of a table or view that is specified in the statement. If no correlation name exists, then the column is null. This column contains the null value if the plan or package was bound using a plan table with fewer than 43 columns. Otherwise, it can contain null if the method that it refers to does not apply.
PAGE_RANGE	CHAR(1) NOT NULL WITH DEFAULT	Indication of whether the table qualifies for page range screening, so that plans scan only the partitions that are needed. **Y** Yes **blank** No

Table 180. Descriptions of columns in PLAN_TABLE (continued)

Column name	Data Type	Description
JOIN_TYPE	CHAR(1) NOT NULL WITH DEFAULT	The type of join: **F** FULL OUTER JOIN **L** LEFT OUTER JOIN **P** Pair-wise join **S** Star join **blank** INNER JOIN or no join RIGHT OUTER JOIN converts to a LEFT OUTER JOIN when you use it, so that JOIN_TYPE contains L.
GROUP_MEMBER	VARCHAR(24) NOT NULL WITH DEFAULT	The member name of the DB2 that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.
IBM_ SERVICE_ DATA	VARCHAR(254) FOR BIT DATA	This column contains values that are for IBM use only.
WHEN_OPTIMIZE	CHAR(1) NOT NULL WITH DEFAULT	When the access path was determined: **blank** At bind time, using a default filter factor for any host variables, parameter markers, or special registers. **B** At bind time, using a default filter factor for any host variables, parameter markers, or special registers; however, the statement is re-optimized at run time using input variable values for input host variables, parameter markers, or special registers. The bind option REOPT(ALWAYS), REOPT(AUTO), or REOPT(ONCE) must be specified for reoptimization to occur. **R** At run time, using input variables for any host variables, parameter markers, or special registers. The bind option REOPT(ALWAYS), REOPT(AUTO), or REOPT(ONCE) must be specified for this to occur.
QBLOCK_TYPE[1]	CHAR(6) NOT NULL WITH DEFAULT	For each query block, an indication of the type of SQL operation that is performed. For the outermost query, this column identifies the statement type.5 on page 2486
BIND_TIME	TIMESTAMP NOT NULL WITH DEFAULT	This column is deprecated. Use EXPLAIN_TIME instead.
OPTHINT	VARCHAR(128) NOT NULL WITH DEFAULT	A string that you use to identify this row as an optimization hint for DB2. DB2 uses this row as input when choosing an access path.

Table 180. Descriptions of columns in PLAN_TABLE (continued)

Column name	Data Type	Description
HINT_USED	VARCHAR(128) NOT NULL WITH DEFAULT	One of the following values: **'APREUSE'** When an access path was successfully reused because the APREUSE option was specified at bind or rebind. '*opthint-value*' When PLAN_TABLE access path hints are used. *opthint-value* is the value of the OPTHINT column for the hint that was used. **'SYSQUERYPLAN** *query-id*' When statement-level access path hints are used. *query-id* is the value of the QUERYID column in the SYSQUERYPLAN catalog table for the hint. **'SYSQUERYSEL** *query-id*' When a predicate selectivity override is used. *query-id* is the value of the QUERYID column of the SYSQUERYSEL catalog table row for the hint. **'EXPLAIN PACKAGE: COPY** *copy-id*' When the row is the result of an EXPLAIN PACKAGE statement. *copy-id* is one of the following values: **0** The current copy. **1** The previous copy. **2** The original copy.
PRIMARY_ ACCESSTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates whether direct row access is attempted first: **D** DB2 tries to use direct row access with a rowid column. If DB2 cannot use direct row access with a rowid column at run time, it uses the access path that is described in the ACCESSTYPE column of PLAN_TABLE. **P** DB2 used data partitioned secondary index and a part-level operation to access the data. **T** The base table or result file is materialized into a work file, and the work file is accessed via sparse index access. If a base table is involved, then ACCESSTYPE indicates how the base table is accessed. **blank** DB2 does not try to use direct row access by using a rowid column or sparse index access for a work file. The value of the ACCESSTYPE column of PLAN_TABLE provides information on the method of accessing the table.

Table 180. Descriptions of columns in PLAN_TABLE (continued)

Column name	Data Type	Description
PARENT_QBLOCKNO	SMALLINT NOT NULL WITH DEFAULT	A number that indicates the QBLOCKNO of the parent query block.
TABLE_TYPE	CHAR(1)	The type of new table: **B** Buffers for SELECT from INSERT, SELECT from UPDATE, SELECT from MERGE, or SELECT from DELETE statement. **C** Common table expression **F** Table function **I** The new table is generated from an IN-LIST predicate. If the IN-LIST predicate is selected as the matching predicate, it will be accessed as an in-memory table. **M** Materialized query table **Q** Temporary intermediate result table (not materialized). For the name of a view or nested table expression, a value of Q indicates that the materialization was virtual and not actual. Materialization can be virtual when the view or nested table expression definition contains a UNION ALL that is not distributed. **R** Recursive common table expression **S** Subquery (correlated or non-correlated) **T** Table **W** Work file The value of the column is null if the query uses GROUP BY, ORDER BY, or DISTINCT, which requires an implicit sort.
TABLE_ENCODE	CHAR(1) NOT NULL WITH DEFAULT	The encoding scheme of the table. The possible values are: **A** ASCII **E** EBCDIC **U** Unicode **M** The table contains multiple CCSID sets
TABLE_SCCSID	SMALLINT NOT NULL WITH DEFAULT	The SBCS CCSID value of the table. If column TABLE_ENCODE is M, the value is 0.
TABLE_MCCSID	SMALLINT NOT NULL WITH DEFAULT	The mixed CCSID value of the table. If the value of the TABLE_ENCODE column is M, the value is 0. If MIXED=NO in the application defaults module, the value is -2.
TABLE_DCCSID	SMALLINT NOT NULL WITH DEFAULT	The DBCS CCSID value of the table. If the value of the TABLE_ENCODE column is M, the value is 0. If MIXED=NO in the application defaults module, the value is -2.

Table 180. Descriptions of columns in PLAN_TABLE (continued)

Column name	Data Type	Description
ROUTINE_ID	INTEGER NOT NULL WITH DEFAULT	The values in this column are for IBM use only.
CTEREF	SMALLINT NOT NULL WITH DEFAULT	If the referenced table is a common table expression, the value is the top-level query block number.
STMTTOKEN	VARCHAR(240)	User-specified statement token.
PARENT_PLANNO	SMALLINT NOT NULL	Corresponds to the plan number in the parent query block where a correlated subquery is invoked. Or, for non-correlated subqueries, corresponds to the plan number in the parent query block that represents the work file for the subquery.
BIND_EXPLAIN_ONLY	CHAR(1) NOT NULL WITH DEFAULT	Identifies whether the row was inserted because a command specified the EXPLAIN(ONLY) option.
SECTNOI	INTEGER NOT NULL WITH DEFAULT	The section number of the statement. The value is taken from the same column in SYSPACKSTMT or SYSSTMT tables and can be used to join tables to reconstruct the access path for the statement. This column is applicable only for static statements. The default value of -1 indicates EXPLAIN information that was captured in Version 9 or earlier.
EXPLAIN_TIME	TIMESTAMP NOT NULL WITH DEFAULT	The time when the EXPLAIN information was captured: **All cached statements** When the statement entered the cache, in the form of a full-precision timestamp value. **Non-cached static statements** When the statement was bound, in the form of a full precision timestamp value. **Non-cached dynamic statements** When EXPLAIN was executed, in the form of a value equivalent to a CHAR(16) representation of the time appended by 4 zeros.
MERGC	CHAR(1) NOT NULL WITH DEFAULT	Indicates whether the composite table is consolidated before the join. **Y** Yes **N** No

Table 180. Descriptions of columns in PLAN_TABLE (continued)

Column name	Data Type	Description	
MERGN	CHAR(1) NOT NULL WITH DEFAULT	Indicates whether the new table is consolidated before the join, or whether access that used a data partitioned secondary index (DPSI) involved a merge operation.	
		Y	Yes, the new table is consolidated before the join.
		N	No, the new table is not consolidated before the join
		D	Access through a DPSI involved a merge operation.
		U	Access through a DPSI that did not involve a merge operation.
SCAN_DIRECTION	CHAR(1)	For index access, the direction of the index scan:	
		F	Forward
		R	Reverse
		blank	Index scan is not used

Table 180. Descriptions of columns in PLAN_TABLE (continued)

Column name	Data Type	Description
EXPANSION_REASON	CHAR(2) NOT NULL WITH DEFAULT	This column applies to only statements that reference archive tables or temporal tables. For other statements, this column is blank. Indicates the effect of the CURRENT TEMPORAL BUSINESS_TIME special register, the CURRENT TEMPORAL SYSTEM_TIME special register, and the SYSIBMADM.GET_ARCHIVE built-in global variable. These items are controlled by the BUSTIMESENSITIVE, SYSTIMESENSITIVE, and ARCHIVESENSITIVE bind options. DB2 implicitly adds certain syntax to the query if one of the following conditions are true: • The SYSIBMADM.GET_ARCHIVE global variable is set to Y and the ARCHIVESENSITIVE bind option is set to YES • The CURRENT TEMPORAL BUSINESS_TIME special register is not null and the BUSTIMESENSITIVE bind option is set to YES • The CURRENT TEMPORAL SYSTEM_TIME special register is not null and the SYSTIMESENSITIVE bind option is set to YES This column can have one of the following values: **A** The query contains implicit query transformation as a result of the SYSIBMADM.GET_ARCHIVE built-in global variable. **B** The query contains implicit query transformation as a result of the CURRENT TEMPORAL BUSINESS_TIME special register. **S** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register. **SB** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register and the CURRENT TEMPORAL BUSINESS_TIME special register. **blank** The query does not contain implicit query transformation.

Notes:

1. For PLAN_TABLE rows in which ACCESSTYPE='A' and QBLOCK_TYPE='SELECT', the values of all other columns except QUERYNO, APPLNAME, and PROGNAME are the default values for those columns.
2. In rows that are used for optimization hints, NULL values in the following columns indicate a hint for no parallelism:

- PARALLELISM_MODE
- ACCESS_PGROUP_ID
- JOIN_PGROUP_ID

3. In rows that are used for optimization hints, NULL values in the following columns indicate a hint for no parallel sort:
   - SORTN_PGROUP_ID
   - SORTC_PGROUP_ID
4. The ACCESSTYPE column contains the following values:

   A The query is sent to an accelerator server.

   DI By an intersection of multiple DOCID lists to return the final DOCID list

   DU By a union of multiple DOCID lists to return the final DOCID list

   DX By an XML index scan on the index that is named in ACCESSNAME to return a DOCID list

   E By direct row access using a row change timestamp column.

   H By hash access. IF an overflow condition occurs, the hash overflow index that is identified by ACCESSCREATOR and ACCESSNAME is used.

   HN By hash access using an IN predicate, or an IN predicate that DB2 generates. If a hash overflow condition occurs, the hash overflow index that is identified in ACCESSCREATOR and ACCESSNAME is used.

   I By an index (identified in ACCESSCREATOR and ACCESSNAME)

   IN By an index scan when the matching predicate contains an IN predicate and the IN-list is accessed through an in-memory table.

   I1 By a one-fetch index scan

   M By a multiple index scan. A row that contains this value might be followed by a row that contains one of the following values:
   - DI
   - DU
   - MH
   - MI
   - MU
   - MX

   MH By the hash overflow index named in ACCESSNAME. A row that contains this value always follows a row that contains M.

   MI By an intersection of multiple indexes. A row that contains this value always follows a row that contains M.

   MU By a union of multiple indexes. A row that contains this value always follows a row that contains M.

   MX By an index scan on the index named in ACCESSNAME. When the access method MX follows the access method DX, DI, or DU, the table is accessed by the DOCID index by using the DOCID list that is returned by DX, DI, or DU. A row that contains this value always follows a row that contains M.

   N One of the following types:
   - By an index scan when the matching predicate contains the IN keyword
   - By an index scan when DB2 rewrites a query using the IN keyword

   O By a work file scan, as a result of a subquery.

   NR Range list access.

   P By a dynamic pair-wise index scan

   R By a table space scan

   RW By a work file scan of the result of a materialized user-defined table function

   V By buffers for an INSERT statement within a SELECT

blank Not applicable to the current row

5. The QBLOCK_TYPE column contains the following values:

SELECT
SELECT

INSERT
INSERT

UPDATE
UPDATE

MERGE
MERGE

DELETE
DELETE

SELUPD
SELECT with FOR UPDATE OF

DELCUR
DELETE WHERE CURRENT OF CURSOR

UPDCUR
UPDATE WHERE CURRENT OF CURSOR

CORSUB
Correlated subselect or fullselect

TRUNCA
TRUNCATE

NCOSUB
Noncorrelated subselect or fullselect

TABLEX
Table expression

TRIGGR
WHEN clause on CREATE TRIGGER

UNION
UNION

UNIONA
UNION ALL

INTERS
INTERSECT

INTERA
INTERSECT ALL

EXCEPT
EXCEPT

EXCEPTA
EXCEPT ALL

PRUNED
DB2 does not generate an access path for the query because the query is guaranteed to qualify zero rows, such as the case of an always-false WHERE clause. For example:`WHERE 0=1`

6. DB2 inserts a value into the REMARKS column at bind or rebind when the EXPLAIN(ONLY) option is specified and reuse or comparison fails for an access path. The value might include the following information:
   - A reason code that corresponds to the reason codes in SQLCODE +395 when reuse fails
   - The name of the unmatched PLAN_TABLE column for which comparison failed
   - A string that identifies that unmatched rows where found
7. DB2 inserts a value into the REMARKS column when selectivity overrides cannot be used for a statement . The value contains a reason code that indicates why the selectivity override was not used. The value might also contain additional diagnostic information.

   The reason code values correspond to SQLCODE +395 reason codes:

   '1'-'41' Indicate that an optimization hint that was generated as part of the extended optimization process cannot be applied. Use only a single selectivity instance.

   '42' Indicates that the structure of the selectivity override is not valid. Generate the selectivity override again.

   '43' The selectivity override cannot be applied because of an unexpected error. If the problem persists, you might need to contact IBM Software Support.

   '44'-'99'
   Indicate that an optimization hint that was generated as part of the extended optimization process cannot be applied. Use only a single selectivity instance.

The PLAN_TABLE_HINT_IX index

The PLAN_TABLE_HINT_IX index improves prepare performance when access path hints are used. This index is required for statement-level access paths and optimization parameters. The PLAN_TABLE_HINT_IX index is optional, although strongly recommended, for PLAN_TABLE access path hints.

The statement that creates the PLAN_TABLE_HINT_IX index is included as part of the DSNTESC member of the SDSNSAMP library. PSPI

Related concepts:

Interpreting data access by using EXPLAIN (DB2 Performance)

Related tasks:

Preparing to influence access paths (DB2 Performance)

Generating visual representations of access plans

Related reference:

“EXPLAIN” on page 1642

Information about one example of an IBM accelerator product

EXPLAIN table changes in Version 11 (DB2 for z/OS What's New?)

DSN_COLDIST_TABLE

The column distribution table contains non-uniform column group statistics that are obtained dynamically by DB2 from non-index leaf pages.

PSPI

Recommendation: Do not manually insert data into system-maintained EXPLAIN tables, and use care when deleting obsolete EXPLAIN table data. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools. Certain optimization tools depend on instances of the various EXPLAIN tables. Be careful not to delete data from or drop instances EXPLAIN tables that are created for these tools.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM
One instance of each EXPLAIN table can be created with the SYSIBM qualifier. DB2 and SQL optimization tools might use these tables. These tables are created when you run job DSNTIJSG when you install or migrate DB2.

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind. They are also populated when you specify EXPLAIN(YES) or EXPLAIN(ONLY) in a BIND or REBIND command. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions

PSPI

The following table shows the descriptions of the columns in the DSN_COLDIST_TABLE table.

Table 181. Descriptions of columns in DSN_COLDIST_TABLE

Column name	Data Type	Description
QUERYNO	INTEGER NOT NULL	A number that identifies the statement that is being explained. The origin of the value depends on the context of the row: **For rows produced by EXPLAIN statements** The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax. **For rows not produced by EXPLAIN statements** DB2 assigns a number that is based on the line number of the SQL statement in the source program. When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in certain rare cases, the value is not guaranteed to be unique. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, if the QUERYNO clause is specified, then its value is used by DB2. Otherwise DB2 assigns a number based on the line number of the SQL statement in the non-inline SQL function, native SQL procedure.
APPLNAME	VARCHAR(128) NOT NULL	The name of the application plan for the row. Applies only to embedded EXPLAIN statements that are executed from a plan or to statements that are explained when binding a plan. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
PROGNAME	VARCHAR(128) NOT NULL	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.

Table 181. Descriptions of columns in DSN_COLDIST_TABLE (continued)

Column name	Data Type	Description
COLLID	VARCHAR(128) NOT NULL	The collection ID: **DSNDYNAMICSQLCACHE** The row originates from the dynamic statement cache **DSNEXPLAINMODEYES** The row originates from an application that specifies YES for the value of the CURRENT EXPLAIN MODE special register. **DSNEXPLAINMODEEXPLAIN** The row originates from an application that specifies EXPLAIN for the value of the CURRENT EXPLAIN MODE special register. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
GROUP_MEMBER	VARCHAR(128) NOT NULL	The member name of the DB2 that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.
SECTNOI	INTEGER NOT NULL	The section number of the statement. The value is taken from the same column in SYSPACKSTMT or SYSSTMT tables and can be used to join tables to reconstruct the access path for the statement. This column is applicable only for static statements. The default value of -1 indicates EXPLAIN information that was captured in Version 9 or earlier.
VERSION	VARCHAR(122) NOT NULL	The version identifier for the package. Applies only to an embedded EXPLAIN statement executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
EXPLAIN_TIME	TIMESTAMP NOT NULL	The time when the EXPLAIN information was captured: **All cached statements** When the statement entered the cache, in the form of a full-precision timestamp value. **Non-cached static statements** When the statement was bound, in the form of a full precision timestamp value. **Non-cached dynamic statements** When EXPLAIN was executed, in the form of a value equivalent to a CHAR(16) representation of the time appended by 4 zeros.

Table 181. Descriptions of columns in DSN_COLDIST_TABLE (continued)

Column name	Data Type	Description
SCHEMA	VARCHAR(128) NOT NULL	The schema of the table that contains the column.
TBNAME	VARCHAR(128) NOT NULL	The name of the table that contains the column.
NAME	VARCHAR(128) NOT NULL	Name of the column. If the value of NUMCOLUMNS is greater than 1, this name identifies the first column name of the set of columns associated with the statistics.
COLVALUE	VARCHAR(2000) NOT NULL FOR BIT DATA	Contains the data of a frequently occurring value in the column. Statistics are not collected for an index on a ROWID column. If the value has a non-character data type, the data might not be printable. This column might contain values that depend on the value of the type column: **TYPE='T'** One of the following values: • 'E3C2C1C3C1D9C4C6' for TBACARDF • 'E3C2C1D5C1C3E3C6' for TBANPAGF • 'E3C2C1D5D7C1C7C6' for TBANACTF **TYPE='L'** 'C3C1E3C6D3C4C3C6' for CATFLDCF **TYPE='P'** One of the following values: • 'D7C3C1D7D5D9E6C6' for PCAPNRWF • 'D7C3C1D7D5D7C7C6' for PCAPNPGF
TYPE	CHAR(1) NOT NULL	The type of statistics: **C** Cardinality **F** Frequent value **H** Histogram **T** Real-time table cardinality **L** Real-time column cardinality (unique index only) **P** real-time partition cardinality

Table 181. Descriptions of columns in DSN_COLDIST_TABLE (continued)

Column name	Data Type	Description
CARDF	FLOAT NOT NULL	For TYPE='C', the number of distinct values for the column group. For TYPE='H', the number of distinct values for the column group in a quantile indicated by the value of the QUANTILENO column. For TYPE='T', a value related to real-time statistics table values that are determined by the COLVALUE column. For TYPE= 'L', a value related to a real-time statistics column value that is determined by the COLVALUE column. The QUANTILENO column contains the column number. The NAME column contains the column name. For TYPE='P' a value related to real-time statistics partition value that is determined by the COLVALUE column. The QUANTILENO column contains the partition number.
COLGROUPCOLNO	VARCHAR(254) NOT NULL FOR BIT DATA	The identity of the set of columns associated with the statistics. If the statistics are only associated with a single column, the field contains a zero length. Otherwise, the field is an array of SMALLINT column numbers with a dimension equal to the value in the NUMCOLUMNS column. This is an updatable column.
NUMCOLUMNS	SMALLINT NOT NULL	Identifies the number of columns associated with the statistics.
FREQUENCYF	FLOAT NOT NULL	The percentage of rows in the table with the value that is specified in the COLVALUE column when the number is multiplied by 100. For example, a value of '1' indicates 100%. A value of '.153' indicates 15.3%.
QUANTILENO	SMALLINT NOT NULL	The ordinary sequence number of a quantile in the whole consecutive value range, from low to high. This column is not updatable. For TYPE= 'L', this column contains the column number. For TYPE='P', the column contains the partition number.
LOWVALUE	VARCHAR(2000) NOT NULL FOR BIT DATA	For TYPE='H', this is the lower bound for the quantile indicated by the value of the QUANTILENO column. Not used if the value of the TYPE column is not 'H'. This column is not updatable.
HIGHVALUE	VARCHAR(2000) NOT NULL FOR BIT DATA	For TYPE='H', this is the higher bound for the quantile indicated by the value of the QUANTILENO column. This column is not used if the value of the TYPE column is not 'H'. This column is not updatable.

Table 181. Descriptions of columns in DSN_COLDIST_TABLE (continued)

Column name	Data Type	Description
EXPANSION_REASON	CHAR(2) NOT NULL WITH DEFAULT	This column applies to only statements that reference archive tables or temporal tables. For other statements, this column is blank. Indicates the effect of the CURRENT TEMPORAL BUSINESS_TIME special register, the CURRENT TEMPORAL SYSTEM_TIME special register, and the SYSIBMADM.GET_ARCHIVE built-in global variable. These items are controlled by the BUSTIMESENSITIVE, SYSTIMESENSITIVE, and ARCHIVESENSITIVE bind options. DB2 implicitly adds certain syntax to the query if one of the following conditions are true: • The SYSIBMADM.GET_ARCHIVE global variable is set to Y and the ARCHIVESENSITIVE bind option is set to YES • The CURRENT TEMPORAL BUSINESS_TIME special register is not null and the BUSTIMESENSITIVE bind option is set to YES • The CURRENT TEMPORAL SYSTEM_TIME special register is not null and the SYSTIMESENSITIVE bind option is set to YES This column can have one of the following values: **A** The query contains implicit query transformation as a result of the SYSIBMADM.GET_ARCHIVE built-in global variable. **B** The query contains implicit query transformation as a result of the CURRENT TEMPORAL BUSINESS_TIME special register. **S** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register. **SB** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register and the CURRENT TEMPORAL BUSINESS_TIME special register. **blank** The query does not contain implicit query transformation.

PSPI

Related concepts:

Dynamic collection of index filtering estimates (DB2 Performance)

DSN_DETCOST_TABLE

The detailed cost table, DSN_DETCOST_TABLE, contains information about detailed cost estimation of the mini-plans in a query.

Recommendation: Do not manually insert data into system-maintained EXPLAIN tables, and use care when deleting obsolete EXPLAIN table data. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools. Certain optimization tools depend on instances of the various EXPLAIN tables. Be careful not to delete data from or drop instances EXPLAIN tables that are created for these tools.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM
: One instance of each EXPLAIN table can be created with the SYSIBM qualifier. DB2 and SQL optimization tools might use these tables. These tables are created when you run job DSNTIJSG when you install or migrate DB2.

userID
: You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind. They are also populated when you specify EXPLAIN(YES) or EXPLAIN(ONLY) in a BIND or REBIND command. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions

The following table describes the columns of DSN_DETCOST_TABLE.

Table 182. DSN_DETCOST_TABLE description

Column name	Data type	Description
QUERYNO	INTEGER NOT NULL	A number that identifies the statement that is being explained. The origin of the value depends on the context of the row: **For rows produced by EXPLAIN statements** The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax. **For rows not produced by EXPLAIN statements** DB2 assigns a number that is based on the line number of the SQL statement in the source program. When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in certain rare cases, the value is not guaranteed to be unique. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, if the QUERYNO clause is specified, then its value is used by DB2. Otherwise DB2 assigns a number based on the line number of the SQL statement in the non-inline SQL function, native SQL procedure.
QBLOCKNO	SMALLINT NOT NULL	A number that identifies each query block within a query. The value of the numbers are not in any particular order, nor are they necessarily consecutive.
APPLNAME	VARCHAR(24) NOT NULL	The name of the application plan for the row. Applies only to embedded EXPLAIN statements that are executed from a plan or to statements that are explained when binding a plan. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
PROGNAME	VARCHAR(128) NOT NULL	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
PLANNO	SMALLINT NOT NULL	The plan number, a number used to identify each mini-plan with a query block.
OPENIO	FLOAT(4) NOT NULL	The Do-at-open IO cost for non-correlated subquery.
OPENCPU	FLOAT(4) NOT NULL	The Do-at-open CPU cost for non-correlated subquery.
OPENCOST	FLOAT(4) NOT NULL	The Do-at-open total cost for non-correlated subquery.
DMIO	FLOAT(4) NOT NULL	IBM internal use only.

Table 182. DSN_DETCOST_TABLE description (continued)

Column name	Data type	Description
DMCPU	FLOAT(4) NOT NULL	IBM internal use only.
DMTOT	FLOAT(4) NOT NULL	IBM internal use only.
SUBQIO	FLOAT(4) NOT NULL	IBM internal use only.
SUBQCOST	FLOAT(4) NOT NULL	IBM internal use only.
BASEIO	FLOAT(4) NOT NULL	IBM internal use only.
BASECPU	FLOAT(4) NOT NULL	IBM internal use only.
BASETOT	FLOAT(4) NOT NULL	IBM internal use only.
ONECOMPROWS	FLOAT(4) NOT NULL	The number of rows qualified after applying local predicates.
IMLEAF	FLOAT(4) NOT NULL	The number of index leaf pages scanned by Data Manager.
IMIO	FLOAT(4) NOT NULL	IBM internal use only.
IMPREFH	CHAR(2) NOT NULL	IBM internal use only.
IMMPRED	INTEGER NOT NULL	IBM internal use only.
IMFF	FLOAT(4) NOT NULL	The filter factor of matching predicates only.
IMSRPRED	INTEGER NOT NULL	IBM internal use only.
IMFFADJ	FLOAT(4) NOT NULL	The filter factor of matching and screening predicates.
IMSCANCST	FLOAT(4) NOT NULL	IBM internal use only.
IMROWCST	FLOAT(4) NOT NULL	IBM internal use only.
IMPAGECST	FLOAT(4) NOT NULL	IBM internal use only.
IMRIDSORT	FLOAT(4) NOT NULL	IBM internal use only.
IMMERGCST	FLOAT(4) NOT NULL	IBM internal use only.
IMCPU	FLOAT(4) NOT NULL	IBM internal use only.
IMTOT	FLOAT(4) NOT NULL	IBM internal use only.
IMSEQNO	SMALLINT NOT NULL	IBM internal use only.

Table 182. DSN_DETCOST_TABLE description (continued)

Column name	Data type	Description
DMPEREFH	FLOAT(4) NOT NULL	IBM internal use only.
DMCLUDIO	FLOAT(4) NOT NULL	IBM internal use only.
DMPREDS	INTEGER NOT NULL	IBM internal use only.
DMSROWS	FLOAT(4) NOT NULL	IBM internal use only.
DMSCANCST	FLOAT(4) NOT NULL	IBM internal use only.
DMCOLS	FLOAT(4) NOT NULL	The number of data manager columns.
DMROWS	FLOAT(4) NOT NULL	The number of data manager rows returned (after all stage 1 predicates are applied).
RDSROWCST	FLOAT(4) NOT NULL	IBM internal use only.
DMPAGECST	FLOAT(4) NOT NULL	IBM internal use only.
DMDATAIO	FLOAT(4) NOT NULL	IBM internal use only.
DMDATAIO	FLOAT(4) NOT NULL	IBM internal use only.
DMDATACPU	FLOAT(4) NOT NULL	IBM internal use only.
DMDATACPU	FLOAT(4) NOT NULL	IBM internal use only.
RDSROW	FLOAT(4) NOT NULL	The number of RDS rows returned (after all stage 1 and stage 2 predicates are applied).
SNCOLS	SMALLINT NOT NULL	The number of columns as sort input for new table.
SNROWS	FLOAT(4) NOT NULL	The number of rows as sort input for new table.
SNRECSZ	INTEGER NOT NULL	The record size for new table.
SNPAGES	FLOAT(4) NOT NULL	The page size for new table.
SNRUNS	FLOAT(4) NOT NULL	The number of runs generated for sort of new table.
SNMERGES	FLOAT(4) NOT NULL	The number of merges needed during sort.
SNIOCOST	FLOAT(4) NOT NULL	IBM internal use only.
SNCPUCOST	FLOAT(4) NOT NULL	IBM internal use only.
SNCOST	FLOAT(4) NOT NULL	IBM internal use only.

Table 182. DSN_DETCOST_TABLE description (continued)

Column name	Data type	Description
SNCSCANIO	FLOAT(4) NOT NULL	IBM internal use only.
SNSCANCPU	FLOAT(4) NOT NULL	IBM internal use only.
SNCCOLS	FLOAT(4) NOT NULL	The number of columns as sort input for Composite table.
SCROWS	FLOAT(4) NOT NULL	The number of rows as sort input for Composite Table.
SCRECSZ	FLOAT(4) NOT NULL	The record size for Composite table.
SCPAGES	FLOAT(4) NOT NULL	The page size for Composite table.
SCRUNS	FLOAT(4) NOT NULL	The number of runs generated during sort of composite.
SCMERGES	FLOAT(4) NOT NULL	The number of merges needed during sort of composite.
SCIOCOST	FLOAT(4) NOT NULL	IBM internal use only.
SCCPUCOST	FLOAT(4) NOT NULL	IBM internal use only.
SCCOST	FLOAT(4) NOT NULL	IBM internal use only.
SCSCANIO	FLOAT(4) NOT NULL	IBM internal use only.
SCSCANCPU	FLOAT(4) NOT NULL	IBM internal use only.
SCSCANCOST	FLOAT(4) NOT NULL	IBM internal use only.
COMPCARD	FLOAT(4) NOT NULL	The total composite cardinality.
COMPIOCOST	FLOAT(4) NOT NULL	IBM internal use only.
COMPCPUCOST	FLOAT(4) NOT NULL	IBM internal use only.
COMPCOST	FLOAT(4) NOT NULL	The total cost.
JOINCOLS	SMALLINT NOT NULL	IBM internal use only.

Table 182. DSN_DETCOST_TABLE description (continued)

Column name	Data type	Description
EXPLAIN_TIME	TIMESTAMP NOT NULL	The time when the EXPLAIN information was captured: **All cached statements** When the statement entered the cache, in the form of a full-precision timestamp value. **Non-cached static statements** When the statement was bound, in the form of a full precision timestamp value. **Non-cached dynamic statements** When EXPLAIN was executed, in the form of a value equivalent to a CHAR(16) representation of the time appended by 4 zeros.
COSTBLK	INTEGER NOT NULL	IBM internal use only.
COSTSTOR	INTEGER NOT NULL	IBM internal use only.
MPBLK	INTEGER NOT NULL	IBM internal use only.
MPSTOR	INTEGER NOT NULL	IBM internal use only.
COMPOSITES	INTEGER NOT NULL	IBM internal use only.
CLIPPED	INTEGER NOT NULL	IBM internal use only.
TABREF	VARCHAR(64) NOT NULL FOR BIT DATA	IBM internal use only.
MAX_COMPOSITES	INTEGER NOT NULL	IBM internal use only.
MAX_STOR	INTEGER NOT NULL	IBM internal use only.
MAX_CPU	INTEGER NOT NULL	IBM internal use only.
MAX_ELAP	INTEGER NOT NULL	IBM internal use only.
TBL_JOINED_THRESH	INTEGER NOT NULL	IBM internal use only.
STOR_USED	INTEGER NOT NULL	IBM internal use only.
CPU_USED	INTEGER NOT NULL	IBM internal use only.
ELAPSED	INTEGER NOT NULL	IBM internal use only.
MIN_CARD_KEEP	FLOAT(4) NOT NULL	IBM internal use only.
MAX_CARD_KEEP	FLOAT(4) NOT NULL	IBM internal use only.
MIN_COST_KEEP	FLOAT(4) NOT NULL	IBM internal use only.

Table 182. DSN_DETCOST_TABLE description (continued)

Column name	Data type	Description
MAX_COST_KEEP	FLOAT(4) NOT NULL	IBM internal use only.
MIN_VALUE_KEEP	FLOAT(4) NOT NULL	IBM internal use only.
MIN_VALUE_CARD_KEEP	FLOAT(4) NOT NULL	IBM internal use only.
MIN_VALUE_COST_KEEP	FLOAT(4) NOT NULL	IBM internal use only.
MIN_CARD_CLIP	FLOAT(4) NOT NULL	IBM internal use only.
MAX_CARD_CLIP	FLOAT(4) NOT NULL	IBM internal use only.
MIN_COST_CLIP	FLOAT(4) NOT NULL	IBM internal use only.
MAX_COST_CLIP	FLOAT(4) NOT NULL	IBM internal use only.
MIN_VALUE_CLIP	FLOAT(4) NOT NULL	IBM internal use only.
MIN_VALUE_CARD_CLIP	FLOAT(4) NOT NULL	IBM internal use only.
MIN_VALUE_COST_CLIP	FLOAT(4) NOT NULL	IBM internal use only.
MAX_VALUE_CLIP	FLOAT(4) NOT NULL	IBM internal use only.
MAX_VALUE_CARD_CLIP	FLOAT(4) NOT NULL	IBM internal use only.
MAX_VALUE_COST_CLIP	FLOAT(4) NOT NULL	IBM internal use only.
GROUP_MEMBER	VARCHAR(24) NOT NULL	The member name of the DB2 that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.
PSEQIOCOST	FLOAT(4) NOT NULL	IBM internal use only.
PSEQIOCOST	FLOAT(4) NOT NULL	IBM internal use only.
PSEQCPUCOST	FLOAT(4) NOT NULL	IBM internal use only.
PSEQCOST	FLOAT(4) NOT NULL	IBM internal use only.
PADJIOCOST	FLOAT(4) NOT NULL	IBM internal use only.
PADJCPUCOST	FLOAT(4) NOT NULL	IBM internal use only.
PADJCOST	FLOAT(4) NOT NULL	IBM internal use only.

Table 182. DSN_DETCOST_TABLE description (continued)

Column name	Data type	Description
UNCERTAINTY	FLOAT(4) NOT NULL WITH DEFAULT	Describes the uncertainty factor of inner table index access. It is aggregated from uncertainty of inner table probing predicates. A larger value indicates a higher uncertainty. 0 indicates no uncertainty or uncertainty not considered.
UNCERTAINTY_1T	FLOAT(4) NOT NULL WITH DEFAULT	Describes the uncertainty factor of ONECOMPROWS column of the table. It is aggregated from all local predicates on the table. A larger value indicates a higher uncertainty. 0 indicates no uncertainty or uncertainty not considered.
SECTNOI	INTEGER NOT NULL WITH DEFAULT	The section number of the statement. The value is taken from the same column in SYSPACKSTMT or SYSSTMT tables and can be used to join tables to reconstruct the access path for the statement. This column is applicable only for static statements. The default value of -1 indicates EXPLAIN information that was captured in Version 9 or earlier.
COLLID	VARCHAR(128) NOT NULL	The collection ID: **DSNDYNAMICSQLCACHE** The row originates from the dynamic statement cache **DSNEXPLAINMODEYES** The row originates from an application that specifies YES for the value of the CURRENT EXPLAIN MODE special register. **DSNEXPLAINMODEEXPLAIN** The row originates from an application that specifies EXPLAIN for the value of the CURRENT EXPLAIN MODE special register. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
VERSION	VARCHAR(128) NOT NULL WITH DEFAULT	The version identifier for the package. Applies only to an embedded EXPLAIN statement executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
IMNP	FLOAT(4) NOT NULL WITH DEFAULT	IBM internal use only.
DMNP	FLOAT(4) NOT NULL WITH DEFAULT	IBM internal use only.
IMJC	FLOAT(4) NOT NULL WITH DEFAULT	IBM internal use only.
IMFC	FLOAT(4) NOT NULL WITH DEFAULT	IBM internal use only.
IMJBC	FLOAT(4) NOT NULL WITH DEFAULT	IBM internal use only.

Table 182. DSN_DETCOST_TABLE description (continued)

Column name	Data type	Description
IMJFC	FLOAT(4) NOT NULL WITH DEFAULT	IBM internal use only.
CRED	INTEGER NOT NULL WITH DEFAULT	IBM internal use only.
IXSCAN_SKIP_DUPS	CHAR(1) NOT NULL WITH DEFAULT 'N'	Whether duplicate index key values are skipped during an index scan. **'Y'** Duplicate key values are skipped. **'N'** Duplicate key values are not skipped.
IXSCAN_SKIP_SCREEN	CHAR(1) NOT NULL WITH DEFAULT 'N'	Whether key ranges that are disqualified by index screening predicates are skipped during an index scan. **'Y'** Disqualified key ranges are skipped. **'N'** Key ranges are not skipped.
EARLY_OUT	CHAR(1) NOT NULL WITH DEFAULT ' '	Whether fetching from the table stops after the first qualified row. **'Y'** Internal fetching stops after the first qualified row **'N'** Internal fetching continues after the first qualified row. **blank** The EXPLAIN information was captured in a previous release, or the EXPLAIN information was captured for a package that was bound in a previous release.

Table 182. DSN_DETCOST_TABLE description (continued)

Column name	Data type	Description
EXPANSION_REASON	CHAR(2) NOT NULL WITH DEFAULT	This column applies to only statements that reference archive tables or temporal tables. For other statements, this column is blank. Indicates the effect of the CURRENT TEMPORAL BUSINESS_TIME special register, the CURRENT TEMPORAL SYSTEM_TIME special register, and the SYSIBMADM.GET_ARCHIVE built-in global variable. These items are controlled by the BUSTIMESENSITIVE, SYSTIMESENSITIVE, and ARCHIVESENSITIVE bind options. DB2 implicitly adds certain syntax to the query if one of the following conditions are true: • The SYSIBMADM.GET_ARCHIVE global variable is set to Y and the ARCHIVESENSITIVE bind option is set to YES • The CURRENT TEMPORAL BUSINESS_TIME special register is not null and the BUSTIMESENSITIVE bind option is set to YES • The CURRENT TEMPORAL SYSTEM_TIME special register is not null and the SYSTIMESENSITIVE bind option is set to YES This column can have one of the following values: **A** The query contains implicit query transformation as a result of the SYSIBMADM.GET_ARCHIVE built-in global variable. **B** The query contains implicit query transformation as a result of the CURRENT TEMPORAL BUSINESS_TIME special register. **S** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register. **SB** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register and the CURRENT TEMPORAL BUSINESS_TIME special register. **blank** The query does not contain implicit query transformation.

PSPI

Related reference:

Information about one example of an IBM accelerator product

DSN_FILTER_TABLE

The filter table, DSN_FILTER_TABLE, contains information about how predicates are used during query processing.

Recommendation: Do not manually insert data into system-maintained EXPLAIN tables, and use care when deleting obsolete EXPLAIN table data. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools. Certain optimization tools depend on instances of the various EXPLAIN tables. Be careful not to delete data from or drop instances EXPLAIN tables that are created for these tools.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM
: One instance of each EXPLAIN table can be created with the SYSIBM qualifier. DB2 and SQL optimization tools might use these tables. These tables are created when you run job DSNTIJSG when you install or migrate DB2.

userID
: You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind. They are also populated when you specify EXPLAIN(YES) or EXPLAIN(ONLY) in a BIND or REBIND command. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions

The following table describes the columns of DSN_FILTER_TABLE.

Table 183. DSN_FILTER_TABLE description

Column name	Data type	Description
QUERYNO	INTEGER NOT NULL	A number that identifies the statement that is being explained. The origin of the value depends on the context of the row: **For rows produced by EXPLAIN statements** The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax. **For rows not produced by EXPLAIN statements** DB2 assigns a number that is based on the line number of the SQL statement in the source program. When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in certain rare cases, the value is not guaranteed to be unique. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, if the QUERYNO clause is specified, then its value is used by DB2. Otherwise DB2 assigns a number based on the line number of the SQL statement in the non-inline SQL function, native SQL procedure.
QBLOCKNO	SMALLINT NOT NULL	A number that identifies each query block within a query. The value of the numbers are not in any particular order, nor are they necessarily consecutive.
PLANNO	SMALLINT	The plan number, a number used to identify each miniplan with a query block.
APPLNAME	VARCHAR(24) NOT NULL	The name of the application plan for the row. Applies only to embedded EXPLAIN statements that are executed from a plan or to statements that are explained when binding a plan. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
PROGNAME	VARCHAR(128) NOT NULL	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.

Table 183. DSN_FILTER_TABLE description (continued)

Column name	Data type	Description
COLLID	VARCHAR(128) NOT NULL WITH DEFAULT	The collection ID: **DSNDYNAMICSQLCACHE** The row originates from the dynamic statement cache **DSNEXPLAINMODEYES** The row originates from an application that specifies YES for the value of the CURRENT EXPLAIN MODE special register. **DSNEXPLAINMODEEXPLAIN** The row originates from an application that specifies EXPLAIN for the value of the CURRENT EXPLAIN MODE special register. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
ORDERNO	INTEGER NOT NULL	The sequence number of evaluation. Indicates the order in which the predicate is applied within each stage
PREDNO	INTEGER NOT NULL	The predicate number, a number used to identify a predicate within a query.
STAGE	CHAR(9) NOT NULL	The processing stage in which the predicate is evaluated: **MATCHING** During the index matching stage. **SCREENING** During the index screening stage. **PAGERANGE** DB2 used page range screening to limit the number of partitions that were accessed to evaluate the predicate in a join context. **STAGE1** During stage 1 processing, after data page access. **STAGE2** During stage 2 processing on the returned data rows.
ORDERCLASS	INTEGER NOT NULL	IBM internal use only.
EXPLAIN_TIME	TIMESTAMP NOT NULL	The time when the EXPLAIN information was captured: **All cached statements** When the statement entered the cache, in the form of a full-precision timestamp value. **Non-cached static statements** When the statement was bound, in the form of a full precision timestamp value. **Non-cached dynamic statements** When EXPLAIN was executed, in the form of a value equivalent to a CHAR(16) representation of the time appended by 4 zeros.
MIXOPSEQNO	SMALLINT NOT NULL	IBM internal use only.
REEVAL	CHAR(1) NOT NULL	IBM internal use only.

Table 183. DSN_FILTER_TABLE description (continued)

Column name	Data type	Description
GROUP_MEMBER	VARCHAR(24) NOT NULL	The member name of the DB2 that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.
SECTNOI	INTEGER NOT NULL WITH DEFAULT	The section number of the statement. The value is taken from the same column in SYSPACKSTMT or SYSSTMT tables and can be used to join tables to reconstruct the access path for the statement. This column is applicable only for static statements. The default value of -1 indicates EXPLAIN information that was captured in Version 9 or earlier.
VERSION	VARCHAR(122) NOT NULL WITH DEFAULT	The version identifier for the package. Applies only to an embedded EXPLAIN statement executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
PUSHDOWN	CHAR(1) NOT NULL WITH DEFAULT	Whether the predicate is pushed down the Index Manager or Data Manager subcomponents for evaluation: **'I'** The Index Manager subcomponent evaluates the predicate. **'D'** The Data Manager subcomponent evaluates the predicate. **blank** The predicate is not pushed down for evaluation.

Table 183. DSN_FILTER_TABLE description (continued)

Column name	Data type	Description
EXPANSION_REASON	CHAR(2) NOT NULL WITH DEFAULT	This column applies to only statements that reference archive tables or temporal tables. For other statements, this column is blank. Indicates the effect of the CURRENT TEMPORAL BUSINESS_TIME special register, the CURRENT TEMPORAL SYSTEM_TIME special register, and the SYSIBMADM.GET_ARCHIVE built-in global variable. These items are controlled by the BUSTIMESENSITIVE, SYSTIMESENSITIVE, and ARCHIVESENSITIVE bind options. DB2 implicitly adds certain syntax to the query if one of the following conditions are true: • The SYSIBMADM.GET_ARCHIVE global variable is set to Y and the ARCHIVESENSITIVE bind option is set to YES • The CURRENT TEMPORAL BUSINESS_TIME special register is not null and the BUSTIMESENSITIVE bind option is set to YES • The CURRENT TEMPORAL SYSTEM_TIME special register is not null and the SYSTIMESENSITIVE bind option is set to YES This column can have one of the following values: **A** The query contains implicit query transformation as a result of the SYSIBMADM.GET_ARCHIVE built-in global variable. **B** The query contains implicit query transformation as a result of the CURRENT TEMPORAL BUSINESS_TIME special register. **S** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register. **SB** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register and the CURRENT TEMPORAL BUSINESS_TIME special register. **blank** The query does not contain implicit query transformation.

PSPI

DSN_FUNCTION_TABLE

The function table, DSN_FUNCTION_TABLE, contains descriptions of functions that are used in specified SQL statements.

PSPI

Recommendation: Do not manually insert data into system-maintained EXPLAIN tables, and use care when deleting obsolete EXPLAIN table data. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools. Certain optimization tools depend on instances of the various EXPLAIN tables. Be careful not to delete data from or drop instances EXPLAIN tables that are created for these tools.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM
One instance of each EXPLAIN table can be created with the SYSIBM qualifier. DB2 and SQL optimization tools might use these tables. These tables are created when you run job DSNTIJSG when you install or migrate DB2.

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind. They are also populated when you specify EXPLAIN(YES) or EXPLAIN(ONLY) in a BIND or REBIND command. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions

PSPI

The following table describes the columns of DSN_FUNCTION_TABLE.

Table 184. Descriptions of columns in DSN_FUNCTION_TABLE

Column name	Data type	Description
QUERYNO	INTEGER NOT NULL WITH DEFAULT	A number that identifies the statement that is being explained. The origin of the value depends on the context of the row: **For rows produced by EXPLAIN statements** The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax. **For rows not produced by EXPLAIN statements** DB2 assigns a number that is based on the line number of the SQL statement in the source program. When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in certain rare cases, the value is not guaranteed to be unique. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, if the QUERYNO clause is specified, then its value is used by DB2. Otherwise DB2 assigns a number based on the line number of the SQL statement in the non-inline SQL function, native SQL procedure.
QBLOCKNO	INTEGER NOT NULL WITH DEFAULT	A number that identifies each query block within a query. The value of the numbers are not in any particular order, nor are they necessarily consecutive.
APPLNAME	VARCHAR(24) NOT NULL WITH DEFAULT	The name of the application plan for the row. Applies only to embedded EXPLAIN statements that are executed from a plan or to statements that are explained when binding a plan. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
PROGNAME	VARCHAR(128) NOT NULL WITH DEFAULT	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
COLLID	VARCHAR(128) NOT NULL WITH DEFAULT	The collection ID: **DSNDYNAMICSQLCACHE** The row originates from the dynamic statement cache **DSNEXPLAINMODEYES** The row originates from an application that specifies YES for the value of the CURRENT EXPLAIN MODE special register. **DSNEXPLAINMODEEXPLAIN** The row originates from an application that specifies EXPLAIN for the value of the CURRENT EXPLAIN MODE special register. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.

Table 184. Descriptions of columns in DSN_FUNCTION_TABLE (continued)

Column name	Data type	Description
GROUP_MEMBER	VARCHAR(24) NOT NULL WITH DEFAULT	The member name of the DB2 that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.
EXPLAIN_TIME	TIMESTAMP NOT NULL WITH DEFAULT	The time when the EXPLAIN information was captured: **All cached statements** When the statement entered the cache, in the form of a full-precision timestamp value. **Non-cached static statements** When the statement was bound, in the form of a full precision timestamp value. **Non-cached dynamic statements** When EXPLAIN was executed, in the form of a value equivalent to a CHAR(16) representation of the time appended by 4 zeros.
SCHEMA_NAME	VARCHAR(128) NOT NULL WITH DEFAULT	The schema name of the function invoked in the explained statement.
FUNCTION_NAME	VARCHAR(128) NOT NULL WITH DEFAULT	The name of the function invoked in the explained statement.
SPEC_FUNC_NAME	VARCHAR(128) NOT NULL WITH DEFAULT	The specific name of the function invoked in the explained statement.
FUNCTION_TYPE	CHAR(2) NOT NULL WITH DEFAULT	The type of function invoked in the explained statement. Possible values are: **CU** Column function **SU** Scalar function **TU** Table function
VIEW_CREATOR	VARCHAR(128) NOT NULL WITH DEFAULT	If the function specified in the FUNCTION_NAME column is referenced in a view definition, the creator of the view. Otherwise, blank.
VIEW_NAME	VARCHAR(128) NOT NULL WITH DEFAULT	If the function specified in the FUNCTION_NAME column is referenced in a view definition, the name of the view. Otherwise, blank.
PATH	VARCHAR(2048) NOT NULL WITH DEFAULT	The value of the SQL path that was used to resolve the schema name of the function.
FUNCTION_TEXT	VARCHAR(1500) NOT NULL WITH DEFAULT	The text of the function reference (the function name and parameters). If the function reference is over 100 bytes, this column contains the first 100 bytes. For functions specified in infix notation, FUNCTION_TEXT contains only the function name. For example, for a function named /, which overloads the SQL divide operator, if the function reference is A/B, FUNCTION_TEXT contains only /.
FUNC_VERSION	VARCHAR(122) NOT NULL WITH DEFAULT	For a version of a non-inline SQL scalar function, this column contains the version identifier. For all other cases, this column contains a zero length string. A version of a non-inline SQL scalar function is defined in the SYSIBM.SYSROUTINES table with ORIGIN='Q', FUNCTION_TYPE='S', INLINE='N', and VERSION column containing the version identifier.

Table 184. Descriptions of columns in DSN_FUNCTION_TABLE (continued)

Column name	Data type	Description
SECURE	CHAR(1) NOT NULL WITH DEFAULT	Whether the user-defined function is secure.
SECTNOI	INTEGER NOT NULL WITH DEFAULT	The section number of the statement. The value is taken from the same column in SYSPACKSTMT or SYSSTMT tables and can be used to join tables to reconstruct the access path for the statement. This column is applicable only for static statements. The default value of -1 indicates EXPLAIN information that was captured in Version 9 or earlier.
VERSION	VARCHAR(122) NOT NULL WITH DEFAULT	The version identifier for the package. Applies only to an embedded EXPLAIN statement executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
EXPANSION_REASON	CHAR(2) NOT NULL WITH DEFAULT	This column applies to only statements that reference archive tables or temporal tables. For other statements, this column is blank. Indicates the effect of the CURRENT TEMPORAL BUSINESS_TIME special register, the CURRENT TEMPORAL SYSTEM_TIME special register, and the SYSIBMADM.GET_ARCHIVE built-in global variable. These items are controlled by the BUSTIMESENSITIVE, SYSTIMESENSITIVE, and ARCHIVESENSITIVE bind options. DB2 implicitly adds certain syntax to the query if one of the following conditions are true: • The SYSIBMADM.GET_ARCHIVE global variable is set to Y and the ARCHIVESENSITIVE bind option is set to YES • The CURRENT TEMPORAL BUSINESS_TIME special register is not null and the BUSTIMESENSITIVE bind option is set to YES • The CURRENT TEMPORAL SYSTEM_TIME special register is not null and the SYSTIMESENSITIVE bind option is set to YES This column can have one of the following values: **A** The query contains implicit query transformation as a result of the SYSIBMADM.GET_ARCHIVE built-in global variable. **B** The query contains implicit query transformation as a result of the CURRENT TEMPORAL BUSINESS_TIME special register. **S** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register. **SB** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register and the CURRENT TEMPORAL BUSINESS_TIME special register. **blank** The query does not contain implicit query transformation.

PSPI

Related tasks:

Checking how DB2 resolves functions by using DSN_FUNCTION_TABLE (DB2 Application programming and SQL)

DSN_KEYTGTDIST_TABLE

The key-target distribution table contains non-uniform index expression statistic that are obtained dynamically by the DB2 optimizer.

Recommendation: Do not manually insert data into system-maintained EXPLAIN tables, and use care when deleting obsolete EXPLAIN table data. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools. Certain optimization tools depend on instances of the various EXPLAIN tables. Be careful not to delete data from or drop instances EXPLAIN tables that are created for these tools.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM
One instance of each EXPLAIN table can be created with the SYSIBM qualifier. DB2 and SQL optimization tools might use these tables. These tables are created when you run job DSNTIJSG when you install or migrate DB2.

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind. They are also populated when you specify EXPLAIN(YES) or EXPLAIN(ONLY) in a BIND or REBIND command. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

COLUMN descriptions

The following table shows the descriptions of the columns in the DSN_KEYTGTDIST_TABLE table.

Table 185. Descriptions of columns in DSN_KEYTGTDIST_TABLE

Column name	Data Type	Description
QUERYNO	INTEGER NOT NULL	A number that identifies the statement that is being explained. The origin of the value depends on the context of the row: **For rows produced by EXPLAIN statements** The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax. **For rows not produced by EXPLAIN statements** DB2 assigns a number that is based on the line number of the SQL statement in the source program. When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in certain rare cases, the value is not guaranteed to be unique. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, if the QUERYNO clause is specified, then its value is used by DB2. Otherwise DB2 assigns a number based on the line number of the SQL statement in the non-inline SQL function, native SQL procedure.
APPLNAME	VARCHAR(128) NOT NULL	The name of the application plan for the row. Applies only to embedded EXPLAIN statements that are executed from a plan or to statements that are explained when binding a plan. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
PROGNAME	VARCHAR(128) NOT NULL	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.

Table 185. Descriptions of columns in DSN_KEYTGTDIST_TABLE (continued)

Column name	Data Type	Description
COLLID	VARCHAR(128) NOT NULL	The collection ID: **DSNDYNAMICSQLCACHE** The row originates from the dynamic statement cache **DSNEXPLAINMODEYES** The row originates from an application that specifies YES for the value of the CURRENT EXPLAIN MODE special register. **DSNEXPLAINMODEEXPLAIN** The row originates from an application that specifies EXPLAIN for the value of the CURRENT EXPLAIN MODE special register. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
GROUP_MEMBER	VARCHAR(128) NOT NULL	The member name of the DB2 that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.
SECTNOI	INTEGER NOT NULL	The section number of the statement. The value is taken from the same column in SYSPACKSTMT or SYSSTMT tables and can be used to join tables to reconstruct the access path for the statement. This column is applicable only for static statements. The default value of -1 indicates EXPLAIN information that was captured in Version 9 or earlier.
VERSION	VARCHAR(122) NOT NULL	The version identifier for the package. Applies only to an embedded EXPLAIN statement executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
EXPLAIN_TIME	TIMESTAMP NOT NULL	The time when the EXPLAIN information was captured: **All cached statements** When the statement entered the cache, in the form of a full-precision timestamp value. **Non-cached static statements** When the statement was bound, in the form of a full precision timestamp value. **Non-cached dynamic statements** When EXPLAIN was executed, in the form of a value equivalent to a CHAR(16) representation of the time appended by 4 zeros.

Table 185. Descriptions of columns in DSN_KEYTGTDIST_TABLE (continued)

Column name	Data Type	Description
IXSCHEMA	VARCHAR(128) NOT NULL	The qualifier of the index.
IXNAME	VARCHAR(128) NOT NULL	The name of the index.
KEYSEQ	VARCHAR(128) NOT NULL	The numeric position of the key-target in the index.
KEYVALUE	VARCHAR(2000) NOT NULL FOR BIT DATA	Contains the data of a frequently occurring value. Statistics are not collected for an index on a ROWID column. If the value has a non-character data type, the data might not be printable. When the value of the TYPE column contains 'I', this column contains one of the following values: • 'C9C4E7C6E4D3D2C6' for IDXFULKF • 'C9C4E7D3C5C1C6C6' for IDXLEAFF • 'C9C4E7D5D3E5D3C6' for IDXNLVLF
TYPE	CHAR(1) NOT NULL	The type of statistics: **C** Cardinality **F** Frequent value **H** Histogram **I** Real-time index statistics
CARDF	FLOAT NOT NULL	For TYPE='C', the number of distinct values for the column group. For TYPE='H', the number of distinct values for the column group in a quantile indicated by the value of the QUANTILENO column. For TYPE='I', a value related to real-time index statistics values determined by the KEYVALUE column.
KEYGROUPKEYNO	VARCHAR(254) NOT NULL FOR BIT DATA	Contains a value that identifies the set of keys that are associated with the statistics. If the statistics are associated with more than a single key, it contains an array of SMALLINT key numbers with a dimension that is equal to the value in NUMKEYS. If the statistics are only associated with a single key, it contains 0.
NUMKEYS	SMALLINT NOT NULL	The number of keys that are associated with the statistics.
FREQUENCYF	FLOAT NOT NULL	The percentage of rows in the table with the value that is specified in the COLVALUE column when the number is multiplied by 100. For example, a value of '1' indicates 100%. A value of '.153' indicates 15.3%.
QUANTILENO	SMALLINT NOT NULL	The ordinary sequence number of a quantile in the whole consecutive value range, from low to high. This column is not updatable
LOWVALUE	VARCHAR(2000) NOT NULL FOR BIT DATA	For TYPE='H', this is the lower bound for the quantile indicated by the value of the QUANTILENO column. Not used if the value of the TYPE column is not 'H'. This column is not updatable.

Table 185. Descriptions of columns in DSN_KEYTGTDIST_TABLE (continued)

Column name	Data Type	Description
HIGHVALUE	VARCHAR(2000) NOT NULL FOR BIT DATA	For TYPE='H', this is the higher bound for the quantile indicated by the value of the QUANTILENO column. This column is not used if the value of the TYPE column is not 'H'. This column is not updatable.
EXPANSION_REASON	CHAR(2) NOT NULL WITH DEFAULT	This column applies to only statements that reference archive tables or temporal tables. For other statements, this column is blank. Indicates the effect of the CURRENT TEMPORAL BUSINESS_TIME special register, the CURRENT TEMPORAL SYSTEM_TIME special register, and the SYSIBMADM.GET_ARCHIVE built-in global variable. These items are controlled by the BUSTIMESENSITIVE, SYSTIMESENSITIVE, and ARCHIVESENSITIVE bind options. DB2 implicitly adds certain syntax to the query if one of the following conditions are true: • The SYSIBMADM.GET_ARCHIVE global variable is set to Y and the ARCHIVESENSITIVE bind option is set to YES • The CURRENT TEMPORAL BUSINESS_TIME special register is not null and the BUSTIMESENSITIVE bind option is set to YES • The CURRENT TEMPORAL SYSTEM_TIME special register is not null and the SYSTIMESENSITIVE bind option is set to YES This column can have one of the following values: **A** The query contains implicit query transformation as a result of the SYSIBMADM.GET_ARCHIVE built-in global variable. **B** The query contains implicit query transformation as a result of the CURRENT TEMPORAL BUSINESS_TIME special register. **S** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register. **SB** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register and the CURRENT TEMPORAL BUSINESS_TIME special register. **blank** The query does not contain implicit query transformation.

PSPI

Related concepts:

Dynamic collection of index filtering estimates (DB2 Performance)

DSN_PGRANGE_TABLE

The page range table, DSN_PGRANGE_TABLE, contains information about qualified partitions for all page range scans in a query.

Recommendation: Do not manually insert data into system-maintained EXPLAIN tables, and use care when deleting obsolete EXPLAIN table data. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools. Certain optimization tools depend on instances of the various EXPLAIN tables. Be careful not to delete data from or drop instances EXPLAIN tables that are created for these tools.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM
One instance of each EXPLAIN table can be created with the SYSIBM qualifier. DB2 and SQL optimization tools might use these tables. These tables are created when you run job DSNTIJSG when you install or migrate DB2.

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind. They are also populated when you specify EXPLAIN(YES) or EXPLAIN(ONLY) in a BIND or REBIND command. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions

The following table describes the columns of DSN_PGRANGE_TABLE.

Table 186. DSN_PGRANGE_TABLE description

Column name	Data type	Description
QUERYNO	INTEGER NOT NULL	A number that identifies the statement that is being explained. The origin of the value depends on the context of the row: **For rows produced by EXPLAIN statements** The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax. **For rows not produced by EXPLAIN statements** DB2 assigns a number that is based on the line number of the SQL statement in the source program. When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in certain rare cases, the value is not guaranteed to be unique. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, if the QUERYNO clause is specified, then its value is used by DB2. Otherwise DB2 assigns a number based on the line number of the SQL statement in the non-inline SQL function, native SQL procedure.
QBLOCKNO	SMALLINT NOT NULL	A number that identifies each query block within a query. The value of the numbers are not in any particular order, nor are they necessarily consecutive.
TABNO	SMALLINT NOT NULL	The table number, a number which uniquely identifies the corresponding table reference within a query.
RANGE	SMALLINT NOT NULL	The sequence number of the current page range.
FIRSTPART	SMALLINT NOT NULL	The starting partition in the current page range.
LASTPART	SMALLINT NOT NULL	The ending partition in the current page range.
NUMPARTS	SMALLINT NOT NULL	The number of partitions in the current page range.
EXPLAIN_TIME	TIMESTAMP NOT NULL	The time when the EXPLAIN information was captured: **All cached statements** When the statement entered the cache, in the form of a full-precision timestamp value. **Non-cached static statements** When the statement was bound, in the form of a full precision timestamp value. **Non-cached dynamic statements** When EXPLAIN was executed, in the form of a value equivalent to a CHAR(16) representation of the time appended by 4 zeros.
GROUP_MEMBER	VARCHAR(24) NOT NULL	The member name of the DB2 that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.

Table 186. DSN_PGRANGE_TABLE description (continued)

Column name	Data type	Description
SECTNOI	INTEGER NOT NULL WITH DEFAULT	The section number of the statement. The value is taken from the same column in SYSPACKSTMT or SYSSTMT tables and can be used to join tables to reconstruct the access path for the statement. This column is applicable only for static statements. The default value of -1 indicates EXPLAIN information that was captured in Version 9 or earlier.
APPLNAME	VARCHAR(24) NOT NULL WITH DEFAULT	The name of the application plan for the row. Applies only to embedded EXPLAIN statements that are executed from a plan or to statements that are explained when binding a plan. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
PROGNAME	VARCHAR(128) NOT NULL WITH DEFAULT	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
COLLID	VARCHAR(128) NOT NULL WITH DEFAULT	The collection ID: **DSNDYNAMICSQLCACHE** The row originates from the dynamic statement cache **DSNEXPLAINMODEYES** The row originates from an application that specifies YES for the value of the CURRENT EXPLAIN MODE special register. **DSNEXPLAINMODEEXPLAIN** The row originates from an application that specifies EXPLAIN for the value of the CURRENT EXPLAIN MODE special register. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
VERSION	VARCHAR(122) NOT NULL WITH DEFAULT	The version identifier for the package. Applies only to an embedded EXPLAIN statement executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.

Table 186. DSN_PGRANGE_TABLE description (continued)

Column name	Data type	Description
EXPANSION_REASON	CHAR(2) NOT NULL WITH DEFAULT	This column applies to only statements that reference archive tables or temporal tables. For other statements, this column is blank. Indicates the effect of the CURRENT TEMPORAL BUSINESS_TIME special register, the CURRENT TEMPORAL SYSTEM_TIME special register, and the SYSIBMADM.GET_ARCHIVE built-in global variable. These items are controlled by the BUSTIMESENSITIVE, SYSTIMESENSITIVE, and ARCHIVESENSITIVE bind options. DB2 implicitly adds certain syntax to the query if one of the following conditions are true: • The SYSIBMADM.GET_ARCHIVE global variable is set to Y and the ARCHIVESENSITIVE bind option is set to YES • The CURRENT TEMPORAL BUSINESS_TIME special register is not null and the BUSTIMESENSITIVE bind option is set to YES • The CURRENT TEMPORAL SYSTEM_TIME special register is not null and the SYSTIMESENSITIVE bind option is set to YES This column can have one of the following values: **A** The query contains implicit query transformation as a result of the SYSIBMADM.GET_ARCHIVE built-in global variable. **B** The query contains implicit query transformation as a result of the CURRENT TEMPORAL BUSINESS_TIME special register. **S** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register. **SB** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register and the CURRENT TEMPORAL BUSINESS_TIME special register. **blank** The query does not contain implicit query transformation.

PSPI

DSN_PGROUP_TABLE

The parallel group table, DSN_PGROUP_TABLE, contains information about the parallel groups in a query.

Recommendation: Do not manually insert data into system-maintained EXPLAIN tables, and use care when deleting obsolete EXPLAIN table data. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools. Certain optimization tools depend on instances of the various EXPLAIN tables. Be careful not to delete data from or drop instances EXPLAIN tables that are created for these tools.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM
: One instance of each EXPLAIN table can be created with the SYSIBM qualifier. DB2 and SQL optimization tools might use these tables. These tables are created when you run job DSNTIJSG when you install or migrate DB2.

userID
: You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind. They are also populated when you specify EXPLAIN(YES) or EXPLAIN(ONLY) in a BIND or REBIND command. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions

The following table describes the columns of DSN_PGROUP_TABLE

Table 187. DSN_PGROUP_TABLE description

Column name	Data type	Description
QUERYNO	INTEGER NOT NULL	A number that identifies the statement that is being explained. The origin of the value depends on the context of the row: **For rows produced by EXPLAIN statements** The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax. **For rows not produced by EXPLAIN statements** DB2 assigns a number that is based on the line number of the SQL statement in the source program. When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in certain rare cases, the value is not guaranteed to be unique. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, if the QUERYNO clause is specified, then its value is used by DB2. Otherwise DB2 assigns a number based on the line number of the SQL statement in the non-inline SQL function, native SQL procedure.
QBLOCKNO	SMALLINT NOT NULL	A number that identifies each query block within a query. The value of the numbers are not in any particular order, nor are they necessarily consecutive.
PLANNAME	VARCHAR(24) NOT NULL	The application plan name.
COLLID	VARCHAR(128) NOT NULL	The collection ID: **DSNDYNAMICSQLCACHE** The row originates from the dynamic statement cache **DSNEXPLAINMODEYES** The row originates from an application that specifies YES for the value of the CURRENT EXPLAIN MODE special register. **DSNEXPLAINMODEEXPLAIN** The row originates from an application that specifies EXPLAIN for the value of the CURRENT EXPLAIN MODE special register. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
PROGNAME	VARCHAR(128) NOT NULL	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.

Table 187. DSN_PGROUP_TABLE description (continued)

Column name	Data type	Description
EXPLAIN_TIME	TIMESTAMP NOT NULL	The time when the EXPLAIN information was captured: **All cached statements** When the statement entered the cache, in the form of a full-precision timestamp value. **Non-cached static statements** When the statement was bound, in the form of a full precision timestamp value. **Non-cached dynamic statements** When EXPLAIN was executed, in the form of a value equivalent to a CHAR(16) representation of the time appended by 4 zeros.
VERSION	VARCHAR(122) NOT NULL	The version identifier for the package. Applies only to an embedded EXPLAIN statement executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
GROUPID	SMALLINT NOT NULL	The parallel group identifier within the current query block.
FIRSTPLAN	SMALLINT NOT NULL	The plan number of the first contributing mini-plan associated within this parallel group.
LASTPLAN	SMALLINT NOT NULL	The plan number of the last mini-plan associated within this parallel group.
CPUCOST	REAL NOT NULL	The estimated total CPU cost of this parallel group in milliseconds.
IOCOST	REAL NOT NULL	The estimated total I/O cost of this parallel group in milliseconds.
BESTTIME	REAL NOT NULL	The estimated elapsed time for each parallel task for this parallel group.
DEGREE	SMALLINT NOT NULL	The degree of parallelism for this parallel group determined at bind time. Max parallelism degree if the Table space is large is 255, otherwise 64.
MODE	CHAR(1) NOT NULL	The parallel mode: **'I'** I/O parallelism **'C'** CPU parallelism **'N'** No parallelism
REASON	SMALLINT NOT NULL	The reason code for downgrading parallelism mode.
LOCALCPU	SMALLINT NOT NULL	The number of CPUs currently online when preparing the query.
TOTALCPU	SMALLINT NOT NULL	The total number of CPUs in Sysplex. LOCALCPU and TOTALCPU are different only for the DB2 coordinator in a Sysplex.
FIRSTBASE	SMALLINT	The table number of the table that partitioning is performed on.
LARGETS	CHAR(1)	'Y' if the TableSpace is large in this group.

Table 187. DSN_PGROUP_TABLE description (continued)

Column name	Data type	Description
PARTKIND	CHAR(1)	The partitioning type: **'L'** Logical partitioning **'P'** Physical partitioning
GROUPTYPE	CHAR(3)	Determines what operations this parallel group contains: table Access, Join, or Sort 'A' 'AJ' 'AJS'
ORDER	CHAR(1)	The ordering requirement of this parallel group : **'N'** No order. Results need no ordering. **'T'** Natural Order. Ordering is required but results already ordered if accessed via index. **'K'** Key Order. Ordering achieved by sort. Results ordered by sort key. This value applies only to parallel sort.
STYLE	CHAR(4)	The Input/Output format style of this parallel group. Blank for IO Parallelism. For other modes: **'RIRO'** Records IN, Records OUT **'WIRO'** Work file IN, Records OUT **'WIWO'** Work file IN, Work file OUT
RANGEKIND	CHAR(1)	The range type: **'K'** Key range **'L'** IN-list elements partitioning **'P'** Page range **'R'** Record range partitioning
NKEYCOLS	SMALLINT	The number of interesting key columns, that is, the number of columns that will participate in the key operation for this parallel group.
LOWBOUND	VARCHAR(40) FOR BIT DATA	The low bound of parallel group.
HIGHBOUND	VARCHAR(40) FOR BIT DATA	The high bound of parallel group.
LOWKEY	VARCHAR(40) FOR BIT DATA	The low key of range if partitioned by key range.
HIGHKEY	VARCHAR(40) FOR BIT DATA	The high key of range if partitioned by key range.
FIRSTPAGE	CHAR(4) FOR BIT DATA	The first page in range if partitioned by page range.
LASTPAGE	CHAR(4) FOR BIT DATA	The last page in range if partitioned by page range.

Table 187. DSN_PGROUP_TABLE description (continued)

Column name	Data type	Description
GROUP_MEMBER	VARCHAR(24) NOT NULL	The member name of the DB2 that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.
HOST_REASON	SMALLINT	IBM internal use only.
PARA_TYPE	CHAR(4)	IBM internal use only.
PART_INNER	CHAR(1)	IBM internal use only.
GRNU_KEYRNG	CHAR(1)	IBM internal use only.
OPEN_KEYRNG	CHAR(1)	IBM internal use only.
APPLNAME	VARCHAR(24) NOT NULL WITH DEFAULT	The name of the application plan for the row. Applies only to embedded EXPLAIN statements that are executed from a plan or to statements that are explained when binding a plan. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
SECTNOI	INTEGER NOT NULL WITH DEFAULT	The section number of the statement. The value is taken from the same column in SYSPACKSTMT or SYSSTMT tables and can be used to join tables to reconstruct the access path for the statement. This column is applicable only for static statements. The default value of -1 indicates EXPLAIN information that was captured in Version 9 or earlier.
STRAW_MODEL	CHAR(1) NOT NULL WITH DEFAULT	IBM internal use only.

Table 187. DSN_PGROUP_TABLE description (continued)

Column name	Data type	Description
EXPANSION_REASON	CHAR(2) NOT NULL WITH DEFAULT	This column applies to only statements that reference archive tables or temporal tables. For other statements, this column is blank. Indicates the effect of the CURRENT TEMPORAL BUSINESS_TIME special register, the CURRENT TEMPORAL SYSTEM_TIME special register, and the SYSIBMADM.GET_ARCHIVE built-in global variable. These items are controlled by the BUSTIMESENSITIVE, SYSTIMESENSITIVE, and ARCHIVESENSITIVE bind options. DB2 implicitly adds certain syntax to the query if one of the following conditions are true: • The SYSIBMADM.GET_ARCHIVE global variable is set to Y and the ARCHIVESENSITIVE bind option is set to YES • The CURRENT TEMPORAL BUSINESS_TIME special register is not null and the BUSTIMESENSITIVE bind option is set to YES • The CURRENT TEMPORAL SYSTEM_TIME special register is not null and the SYSTIMESENSITIVE bind option is set to YES This column can have one of the following values: **A** The query contains implicit query transformation as a result of the SYSIBMADM.GET_ARCHIVE built-in global variable. **B** The query contains implicit query transformation as a result of the CURRENT TEMPORAL BUSINESS_TIME special register. **S** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register. **SB** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register and the CURRENT TEMPORAL BUSINESS_TIME special register. **blank** The query does not contain implicit query transformation.

PSPI

DSN_PREDICAT_TABLE

The predicate table, DSN_PREDICAT_TABLE, contains information about all of the predicates in a query. It is also used as input when you issue a BIND QUERY command to override predicate selectivities for matching SQL statements.

Recommendation: Do not manually insert data into system-maintained EXPLAIN tables, and use care when deleting obsolete EXPLAIN table data. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools. Certain optimization tools depend on instances of the various EXPLAIN tables. Be careful not to delete data from or drop instances EXPLAIN tables that are created for these tools.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM
One instance of each EXPLAIN table can be created with the SYSIBM qualifier. DB2 and SQL optimization tools might use these tables. These tables are created when you run job DSNTIJSG when you install or migrate DB2.

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind. They are also populated when you specify EXPLAIN(YES) or EXPLAIN(ONLY) in a BIND or REBIND command. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions

The following table describes the columns of the DSN_PREDICAT_TABLE

Table 188. DSN_PREDICAT_TABLE description

Column name	Data type	Description
QUERYNO	INTEGER NOT NULL	A number that identifies the statement that is being explained. The origin of the value depends on the context of the row: **For rows produced by EXPLAIN statements** The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax. **For rows not produced by EXPLAIN statements** DB2 assigns a number that is based on the line number of the SQL statement in the source program. When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in certain rare cases, the value is not guaranteed to be unique. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, if the QUERYNO clause is specified, then its value is used by DB2. Otherwise DB2 assigns a number based on the line number of the SQL statement in the non-inline SQL function, native SQL procedure.
QBLOCKNO	SMALLINT NOT NULL	A number that identifies each query block within a query. The value of the numbers are not in any particular order, nor are they necessarily consecutive.
APPLNAME	VARCHAR(24) NOT NULL	The name of the application plan for the row. Applies only to embedded EXPLAIN statements that are executed from a plan or to statements that are explained when binding a plan. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
PROGNAME	VARCHAR(128) NOT NULL	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
PREDNO	INTEGER NOT NULL	The predicate number, a number used to identify a predicate within a query.

Table 188. DSN_PREDICAT_TABLE description (continued)

Column name	Data type	Description
TYPE	CHAR(8) NOT NULL	A string used to indicate the type or the operation of the predicate. The possible values are: • 'AND' • 'BETWEEN' • 'EQUAL' • 'EXISTS • 'COMPOUND' • 'HAVING' • 'IN' • 'LIKE' • 'NOT LIKE' • 'NOTEXIST' • 'OTHERS' • 'OR' • 'RANGE' • 'SUBQUERY' • 'XEXISTS' • 'NXEXISTS'
LEFT_HAND_SIDE	VARCHAR(128) NOT NULL	Describes the left side of the predicate. If the left side of the predicate is a table column, this value indicates the name of that column. Other possible values are: • 'VALUE' • 'COLEXP' • 'NONCOLEXP' • 'CORSUB' • 'NONCORSUB' • 'SUBQUERY' • 'EXPRESSION' • Blanks
LEFT_HAND_PNO	INTEGER NOT NULL	If the predicate is a compound predicate (AND/OR), then this column indicates the first child predicate. However, this column is not reliable when the predicate tree consolidation happens. Use PARENT_PNO instead to reconstruct the predicate tree.
LHS_TABNO	SMALLINT NOT NULL	If the left side of the predicate is a table column or a column expression in an expression-based index, then this column indicates a number which uniquely identifies the corresponding table reference within a query.
LHS_QBNO	SMALLINT NOT NULL	If the left side of the predicate is a table column or a column expression in expression-based index, then this column indicates a number which uniquely identifies the corresponding query block within a query.

Table 188. DSN_PREDICAT_TABLE description (continued)

Column name	Data type	Description
RIGHT_HAND_SIDE	VARCHAR(128) NOT NULL	Describes the right side of the predicate. If the right side of the predicate is a table column, this value column indicates the column name. Other possible values are: • 'VALUE' • 'COLEXP' • 'NONCOLEXP' • 'CORSUB' • 'NONCORSUB' • 'SUBQUERY' • 'EXPRESSION' • Blanks
RIGHT_HAND_PNO	INTEGER NOT NULL	If the predicate is a compound predicate (AND/OR), then this column indicates the second child predicate. However, this column is not reliable when the predicate tree consolidation happens. Use PARENT_PNO instead to reconstruct the predicate tree.
RHS_TABNO	CHAR(1) NOT NULL	If the right side of the predicate is a table column or a column expression in an index on expression, then this column indicates a number which uniquely identifies the corresponding table reference within a query.
RHS_QBNO	CHAR(1) NOT NULL	If the right side of the predicate is a subquery or a column expression in an expression-based index, then this column indicates a number which uniquely identifies the corresponding query block within a query.
FILTER_FACTOR	FLOAT NOT NULL	The estimated filter factor.
BOOLEAN_TERM	CHAR(1) NOT NULL	Whether this predicate can be used to determine the truth value of the whole WHERE clause.
SEARCHARG	CHAR(1) NOT NULL	Whether this predicate can be processed by data manager (DM). If it is not, then the relational data service (RDS) needs to be used to take care of it, which is more costly.
JOIN	CHAR(1) NOT NULL	Whether the predicate can be used as a simple join predicate between two tables.
AFTER_JOIN	CHAR(1) NOT NULL	Indicates the predicate evaluation phase: **'A'** After join **'D'** During join **blank** Not applicable

Table 188. DSN_PREDICAT_TABLE description (continued)

Column name	Data type	Description
ADDED_PRED	CHAR(1) NOT NULL	Whether the predicate is generated by DB2, and the reason why the predicate is added: **blank** DB2 did not add the predicate. **'B'** For bubble up. **'C'** For correlation. **'J'** For join. **'K'** For LIKE for expression-based index. **'L'** For localization. **'P'** For push down. **'R'** For page range. **'S'** For simplification. **'T'** For transitive closure.
REDUNDANT_PRED	CHAR(1) NOT NULL	Whether it is a redundant predicate, which means evaluation of other predicates in the query already determines the result that the predicate provides.
DIRECT_ACCESS	CHAR(1) NOT NULL	Whether the predicate is direct access, which means one can navigate directly to the row through ROWID.
KEYFIELD	CHAR(1) NOT NULL	Whether the predicate includes the index key column of the involved table for all applicable indexes considered by DB2.
EXPLAIN_TIME	TIMESTAMP NOT NULL	The time when the EXPLAIN information was captured: **All cached statements** When the statement entered the cache, in the form of a full-precision timestamp value. **Non-cached static statements** When the statement was bound, in the form of a full precision timestamp value. **Non-cached dynamic statements** When EXPLAIN was executed, in the form of a value equivalent to a CHAR(16) representation of the time appended by 4 zeros.
CATEGORY	SMALLINT NOT NULL	IBM internal use only.
CATEGORY_B	SMALLINT NOT NULL	IBM internal use only.
TEXT	VARCHAR(2000) NOT NULL	The text of the transformed predicate text. If the text of the predicate contains more than 2000 characters, it is truncated.
PRED_ENCODE	CHAR(1) NOT NULL WITH DEFAULT	IBM internal use only.
PRED_CCSID	SMALLINT NOT NULL WITH DEFAULT	IBM internal use only.
PRED_MCCSID	SMALLINT NOT NULL WITH DEFAULT	IBM internal use only.

Table 188. DSN_PREDICAT_TABLE description (continued)

Column name	Data type	Description
MARKER	CHAR(1) NOT NULL WITH DEFAULT	Whether this predicate includes host variables, parameter markers, or special registers.
PARENT_PNO	INTEGER NOT NULL	The parent predicate number. If this predicate is a root predicate within a query block, then this column is 0.
NEGATION	CHAR(1) NOT NULL	Whether this predicate is negated via NOT.
LITERALS	VARCHAR(128) NOT NULL	This column indicates the literal value or literal values separated by colon symbols.
CLAUSE	CHAR(8) NOT NULL	The clause where the predicate exists: **'HAVING '** The HAVING clause **'ON '** The ON clause **'WHERE '** The WHERE clause **SELECT** The SELECT clause
GROUP_MEMBER	VARCHAR(24) NOT NULL	The member name of the DB2 that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.
ORIGIN	CHAR(1) NOT NULL WITH DEFAULT	Indicates the origin of the predicate. **Blank** Generated by DB2 **C** Column mask **R** Row permission **U** Specified by the user
UNCERTAINTY	FLOAT(4) NOT NULL WITH DEFAULT	Describes the uncertainty factor of a predicate's estimated filter factor. A bigger value indicates a higher degree of uncertainty. Value zero indicates no uncertainty or uncertainty not considered.
SECTNOI	INTEGER NOT NULL WITH DEFAULT	The section number of the statement. The value is taken from the same column in SYSPACKSTMT or SYSSTMT tables and can be used to join tables to reconstruct the access path for the statement. This column is applicable only for static statements. The default value of -1 indicates EXPLAIN information that was captured in Version 9 or earlier.
COLLID	VARCHAR(128) NOT NULL WITH DEFAULT	The collection ID: **DSNDYNAMICSQLCACHE** The row originates from the dynamic statement cache **DSNEXPLAINMODEYES** The row originates from an application that specifies YES for the value of the CURRENT EXPLAIN MODE special register. **DSNEXPLAINMODEEXPLAIN** The row originates from an application that specifies EXPLAIN for the value of the CURRENT EXPLAIN MODE special register. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.

Table 188. DSN_PREDICAT_TABLE description (continued)

Column name	Data type	Description
VERSION	VARCHAR(122) NOT NULL WITH DEFAULT	The version identifier for the package. Applies only to an embedded EXPLAIN statement executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
EXPANSION_REASON	CHAR(2) NOT NULL WITH DEFAULT	This column applies to only statements that reference archive tables or temporal tables. For other statements, this column is blank. Indicates the effect of the CURRENT TEMPORAL BUSINESS_TIME special register, the CURRENT TEMPORAL SYSTEM_TIME special register, and the SYSIBMADM.GET_ARCHIVE built-in global variable. These items are controlled by the BUSTIMESENSITIVE, SYSTIMESENSITIVE, and ARCHIVESENSITIVE bind options. DB2 implicitly adds certain syntax to the query if one of the following conditions are true: • The SYSIBMADM.GET_ARCHIVE global variable is set to Y and the ARCHIVESENSITIVE bind option is set to YES • The CURRENT TEMPORAL BUSINESS_TIME special register is not null and the BUSTIMESENSITIVE bind option is set to YES • The CURRENT TEMPORAL SYSTEM_TIME special register is not null and the SYSTIMESENSITIVE bind option is set to YES This column can have one of the following values: **A** The query contains implicit query transformation as a result of the SYSIBMADM.GET_ARCHIVE built-in global variable. **B** The query contains implicit query transformation as a result of the CURRENT TEMPORAL BUSINESS_TIME special register. **S** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register. **SB** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register and the CURRENT TEMPORAL BUSINESS_TIME special register. **blank** The query does not contain implicit query transformation.

PSPI

Related concepts:

"Predicates" on page 296

Predicates and access path selection (DB2 Performance)

Related tasks:

Overriding predicate selectivities at the statement level (DB2 Performance)

Related reference:

Tables for influencing access path selection (DB2 Performance)

BIND QUERY (DSN) (DB2 Commands)

DSN_PREDICATE_SELECTIVITY table

The predicate selectivity table contains information about the selectivity of predicates that are used for access path selection. It is used as an input table for the BIND QUERY command when selectivity overrides are specified.

PSPI

When selectivity overrides are not specified, or specified selectivity overrides cannot not be used by DB2, the DSN_PREDICATE_SELECTIVITY table contains one row for each predicate in DSN_PREDICAT_TABLE that is used for access path selection. These rows contain ASSUMPTION='NORMAL' values. DSN_PREDICATE_SELECTIVITY does not contain rows from DSN_PREDICAT_TABLE for predicates that are not used for access path selection.

When selectivity overrides are specified and used by DB2, this table also contains one row for each selectivity override that was used. These rows contain ASSUMPTION='OVERRIDE' values.

Additionally, if the sum of the weights for all specified selectivity override instances is less than one, this table contains one row for each predicate in DSN_PREDICAT_TABLE that is used for access path selection. These rows contain ASSUMPTION='NORMAL' values and WEIGHT values equal to one minus the sum of the specified override weight values.

Recommendation: Do not manually insert data into system-maintained EXPLAIN tables, and use care when deleting obsolete EXPLAIN table data. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools. Certain optimization tools depend on instances of the various EXPLAIN tables. Be careful not to delete data from or drop instances EXPLAIN tables that are created for these tools.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM
One instance of each EXPLAIN table can be created with the SYSIBM qualifier. DB2 and SQL optimization tools might use these tables. These tables are created when you run job DSNTIJSG when you install or migrate DB2.

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind. They are also populated when you specify EXPLAIN(YES) or EXPLAIN(ONLY) in a BIND or REBIND command. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions

Your subsystem or data sharing group can contain more than one of these tables, including a table with the qualifier SYSIBM, a table with the qualifier DB2OSCA, and additional tables that are qualified by user IDs.

The following table shows the descriptions of the columns in the DSN_PREDICATE_SELECTIVITY table.

Table 189. Descriptions of columns in the DSN_PREDICATE_SELECTIVITY table

Column name	Data Type	Description
QUERYNO	INTEGER NOT NULL	A number that identifies the statement that is being explained. The origin of the value depends on the context of the row: **For rows produced by EXPLAIN statements** The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax. **For rows not produced by EXPLAIN statements** DB2 assigns a number that is based on the line number of the SQL statement in the source program. When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in certain rare cases, the value is not guaranteed to be unique. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, if the QUERYNO clause is specified, then its value is used by DB2. Otherwise DB2 assigns a number based on the line number of the SQL statement in the non-inline SQL function, native SQL procedure.
QBLOCKNO	SMALLINT NOT NULL	A number that identifies each query block within a query. The value of the numbers are not in any particular order, nor are they necessarily consecutive.
APPLNAME	VARCHAR(24) NOT NULL	The name of the application plan for the row. Applies only to embedded EXPLAIN statements that are executed from a plan or to statements that are explained when binding a plan. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.

Table 189. Descriptions of columns in the DSN_PREDICATE_SELECTIVITY table (continued)

Column name	Data Type	Description
PROGNAME	VARCHAR(128) NOT NULL	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
SECTNOI	INTEGER NOT NULL WITH DEFAULT	The section number of the statement. The value is taken from the same column in SYSPACKSTMT or SYSSTMT tables and can be used to join tables to reconstruct the access path for the statement. This column is applicable only for static statements. The default value of -1 indicates EXPLAIN information that was captured in Version 9 or earlier.
COLLID	VARCHAR(128) NOT NULL	The collection ID: **DSNDYNAMICSQLCACHE** The row originates from the dynamic statement cache **DSNEXPLAINMODEYES** The row originates from an application that specifies YES for the value of the CURRENT EXPLAIN MODE special register. **DSNEXPLAINMODEEXPLAIN** The row originates from an application that specifies EXPLAIN for the value of the CURRENT EXPLAIN MODE special register. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
VERSION	VARCHAR(122) NOT NULL	The version identifier for the package. Applies only to an embedded EXPLAIN statement executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
PREDNO	INTEGER NOT NULL	The predicate number, a number used to identify a specific predicate within a query.
INSTANCE	SMALLINT NOT NULL	The selectivity instance. Used to group related selectivities.
SELECTIVITY	FLOAT NOT NULL	The selectivity estimate.

Table 189. Descriptions of columns in the DSN_PREDICATE_SELECTIVITY table (continued)

Column name	Data Type	Description
WEIGHT	FLOAT(4) NOT NULL	The percentage of executions that have the specified selectivity. For example, a value of 0.25 means that 25% of the time when query is executed it has this selectivity.
ASSUMPTION	VARCHAR(128) NOT NULL	Indicates how the selectivity was estimated, or is used. One of the following values: **'NORMAL'** Selectivity is estimated by using the normal selectivity assumptions. **'OVERRIDE'** Selectivity is based on an override.
INSERT_TIME	TIMESTAMP NOT NULL GENERATED ALWAYS FOR EACH ROW ON UPDATE AS ROW CHANGE TIMESTAMP	The time when the row was inserted or updated.
EXPLAIN_TIME	TIMESTAMP	The time when the EXPLAIN information was captured: **All cached statements** When the statement entered the cache, in the form of a full-precision timestamp value. **Non-cached static statements** When the statement was bound, in the form of a full precision timestamp value. **Non-cached dynamic statements** When EXPLAIN was executed, in the form of a value equivalent to a CHAR(16) representation of the time appended by 4 zeros.
REMARKS	VARCHAR(762)	IBM internal use only.

Table 189. Descriptions of columns in the DSN_PREDICATE_SELECTIVITY table (continued)

Column name	Data Type	Description
EXPANSION_REASON	CHAR(2) NOT NULL WITH DEFAULT	This column applies to only statements that reference archive tables or temporal tables. For other statements, this column is blank. Indicates the effect of the CURRENT TEMPORAL BUSINESS_TIME special register, the CURRENT TEMPORAL SYSTEM_TIME special register, and the SYSIBMADM.GET_ARCHIVE built-in global variable. These items are controlled by the BUSTIMESENSITIVE, SYSTIMESENSITIVE, and ARCHIVESENSITIVE bind options. DB2 implicitly adds certain syntax to the query if one of the following conditions are true: • The SYSIBMADM.GET_ARCHIVE global variable is set to Y and the ARCHIVESENSITIVE bind option is set to YES • The CURRENT TEMPORAL BUSINESS_TIME special register is not null and the BUSTIMESENSITIVE bind option is set to YES • The CURRENT TEMPORAL SYSTEM_TIME special register is not null and the SYSTIMESENSITIVE bind option is set to YES This column can have one of the following values: **A** The query contains implicit query transformation as a result of the SYSIBMADM.GET_ARCHIVE built-in global variable. **B** The query contains implicit query transformation as a result of the CURRENT TEMPORAL BUSINESS_TIME special register. **S** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register. **SB** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register and the CURRENT TEMPORAL BUSINESS_TIME special register. **blank** The query does not contain implicit query transformation.

PSPI

Related tasks:

- Overriding predicate selectivities at the statement level (DB2 Performance)

Related reference:

- BIND QUERY (DSN) (DB2 Commands)
- DSN_PREDICAT_TABLE (DB2 Performance)
- Tables for influencing access path selection (DB2 Performance)

DSN_PTASK_TABLE

The parallel tasks table, DSN_PTASK_TABLE, contains information about all of the parallel tasks in a query.

Recommendation: Do not manually insert data into system-maintained EXPLAIN tables, and use care when deleting obsolete EXPLAIN table data. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools. Certain optimization tools depend on instances of the various EXPLAIN tables. Be careful not to delete data from or drop instances EXPLAIN tables that are created for these tools.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM
One instance of each EXPLAIN table can be created with the SYSIBM qualifier. DB2 and SQL optimization tools might use these tables. These tables are created when you run job DSNTIJSG when you install or migrate DB2.

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind. They are also populated when you specify EXPLAIN(YES) or EXPLAIN(ONLY) in a BIND or REBIND command. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions

The following table describes the columns of DSN_PTASK_TABLE.

Table 190. DSN_PTASK_TABLE description

Column name	Data type	Description
QUERYNO	INTEGER NOT NULL	A number that identifies the statement that is being explained. The origin of the value depends on the context of the row: **For rows produced by EXPLAIN statements** The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax. **For rows not produced by EXPLAIN statements** DB2 assigns a number that is based on the line number of the SQL statement in the source program. When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in certain rare cases, the value is not guaranteed to be unique. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, if the QUERYNO clause is specified, then its value is used by DB2. Otherwise DB2 assigns a number based on the line number of the SQL statement in the non-inline SQL function, native SQL procedure.
QBLOCKNO	SMALLINT NOT NULL	A number that identifies each query block within a query. The value of the numbers are not in any particular order, nor are they necessarily consecutive.
PGDNO	SMALLINT NOT NULL	The parallel group identifier within the current query block. This value corresponds to the value of the GROUPID column in DSN_PGROUP_TABLE table rows.
APPLNAME	VARCHAR(24) NOT NULL	The name of the application plan for the row. Applies only to embedded EXPLAIN statements that are executed from a plan or to statements that are explained when binding a plan. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
PROGNAME	VARCHAR(128) NOT NULL	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
LPTNO	SMALLINT NOT NULL	The parallel task number.
KEYCOLID	SMALLINT	The key column ID (KEY range only).
DPSI	CHAR(1) NOT NULL	Indicates if a data partition secondary index (DPSI) is used.
LPTLOKEY	VARCHAR(40) FOR BIT DATA	The low key value for this key column for this parallel task (KEY range only).
LPTHIKEY	VARCHAR(40) FOR BIT DATA	The high key value for this key column for this parallel task (KEY range only).

Table 190. DSN_PTASK_TABLE description (continued)

Column name	Data type	Description
LPTLOPAG	CHAR(4) FOR BIT DATA	The low page information if partitioned by page range.
LPTLHIPAG	CHAR(4) FOR BIT DATA	The high page information if partitioned by page range.
LPTLOPG[1]	CHAR(4) FOR BIT DATA	The lower bound page number for this parallel task (Page range or DPSI enabled only).
LPTHIPG[1]	CHAR(4) FOR BIT DATA	The upper bound page number for this parallel task (Page range or DPSI enabled only).
LPTLOPT[1]	SMALLINT	The lower bound partition number for this parallel task (Page range or DPSI enabled only).
LPTHIPT[1]	SMALLINT	The upper bound partition number for this parallel task (Page range or DPSI enabled only).
KEYCOLDT	SMALLINT	The data type for this key column (KEY range only).
KEYCOLPREC	SMALLINT	The precision/length for this key column (KEY range only).
KEYCOLSCAL	SMALLINT	The scale for this key column (KEY range with Decimal datatype only).
EXPLAIN_TIME	TIMESTAMP NOT NULL	The time when the EXPLAIN information was captured: **All cached statements** When the statement entered the cache, in the form of a full-precision timestamp value. **Non-cached static statements** When the statement was bound, in the form of a full precision timestamp value. **Non-cached dynamic statements** When EXPLAIN was executed, in the form of a value equivalent to a CHAR(16) representation of the time appended by 4 zeros.
GROUP_MEMBER	VARCHAR(24) NOT NULL	The member name of the DB2 that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.
SECTNOI	INTEGER NOT NULL WITH DEFAULT WITH DEFAULT	The section number of the statement. The value is taken from the same column in SYSPACKSTMT or SYSSTMT tables and can be used to join tables to reconstruct the access path for the statement. This column is applicable only for static statements. The default value of -1 indicates EXPLAIN information that was captured in Version 9 or earlier.

Table 190. DSN_PTASK_TABLE description (continued)

Column name	Data type	Description
COLLID	VARCHAR(128) NOT NULL WITH DEFAULT	The collection ID: **DSNDYNAMICSQLCACHE** The row originates from the dynamic statement cache **DSNEXPLAINMODEYES** The row originates from an application that specifies YES for the value of the CURRENT EXPLAIN MODE special register. **DSNEXPLAINMODEEXPLAIN** The row originates from an application that specifies EXPLAIN for the value of the CURRENT EXPLAIN MODE special register. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
VERSION	VARCHAR(122) NOT NULL WITH DEFAULT	The version identifier for the package. Applies only to an embedded EXPLAIN statement executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.

Table 190. DSN_PTASK_TABLE description (continued)

Column name	Data type	Description
EXPANSION_REASON	CHAR(2) NOT NULL WITH DEFAULT	This column applies to only statements that reference archive tables or temporal tables. For other statements, this column is blank. Indicates the effect of the CURRENT TEMPORAL BUSINESS_TIME special register, the CURRENT TEMPORAL SYSTEM_TIME special register, and the SYSIBMADM.GET_ARCHIVE built-in global variable. These items are controlled by the BUSTIMESENSITIVE, SYSTIMESENSITIVE, and ARCHIVESENSITIVE bind options. DB2 implicitly adds certain syntax to the query if one of the following conditions are true: • The SYSIBMADM.GET_ARCHIVE global variable is set to Y and the ARCHIVESENSITIVE bind option is set to YES • The CURRENT TEMPORAL BUSINESS_TIME special register is not null and the BUSTIMESENSITIVE bind option is set to YES • The CURRENT TEMPORAL SYSTEM_TIME special register is not null and the SYSTIMESENSITIVE bind option is set to YES This column can have one of the following values: **A** The query contains implicit query transformation as a result of the SYSIBMADM.GET_ARCHIVE built-in global variable. **B** The query contains implicit query transformation as a result of the CURRENT TEMPORAL BUSINESS_TIME special register. **S** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register. **SB** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register and the CURRENT TEMPORAL BUSINESS_TIME special register. **blank** The query does not contain implicit query transformation.

Notes:

1. The name of these columns originally contained the # symbol as the last character in the names. However, the names that contain these characters are obsolete and are no longer supported.

PSPI

DSN_QUERYINFO_TABLE

The query information table, DSN_QUERYINFO_TABLE, contains information about the eligibility of query blocks for automatic query rewrite, information about the materialized query tables that are considered for eligible query blocks, reasons why ineligible query blocks are not eligible, and information about acceleration of query blocks.

PSPI

Recommendation: Do not manually insert data into system-maintained EXPLAIN tables, and use care when deleting obsolete EXPLAIN table data. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools. Certain optimization tools depend on instances of the various EXPLAIN tables. Be careful not to delete data from or drop instances EXPLAIN tables that are created for these tools.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind, or rebind, a plan or package with the EXPLAIN(YES) option. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions

Table 191. Descriptions of columns in DSN_QUERYINFO_TABLE

Column name	Data type	Description
QUERYNO	INTEGER NOT NULL	A number that identifies the statement that is being explained. The origin of the value depends on the context of the row: **For rows produced by EXPLAIN statements** The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax. **For rows not produced by EXPLAIN statements** DB2 assigns a number that is based on the line number of the SQL statement in the source program. When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in certain rare cases, the value is not guaranteed to be unique. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, if the QUERYNO clause is specified, then its value is used by DB2. Otherwise DB2 assigns a number based on the line number of the SQL statement in the non-inline SQL function, native SQL procedure.
QBLOCKNO	SMALLINT NOT NULL	A number that identifies each query block within a query. The value of the numbers are not in any particular order, nor are they necessarily consecutive.
QINAME1	VARCHAR(128) NOT NULL WITH DEFAULT	When TYPE='A': • When REASON_CODE=0, this value is the name of the accelerator server to which the query is sent. • When REASON_CODE<>0, the query was not sent to an accelerator server. The REASON_CODE value indicates why the query was not sent to the accelerator server.
QINAME2	VARCHAR(128) NOT NULL WITH DEFAULT	When TYPE='A' and REASON_CODE=0, this value is the name of the location name of the accelerator server to which the query is sent.
APPLNAME	VARCHAR(24) NOT NULL	The name of the application plan for the row. Applies only to embedded EXPLAIN statements that are executed from a plan or to statements that are explained when binding a plan. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
PROGNAME	VARCHAR(128) NOT NULL	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.

Table 191. Descriptions of columns in DSN_QUERYINFO_TABLE (continued)

Column name	Data type	Description
VERSION	VARCHAR(122) NOT NULL	The version identifier for the package. Applies only to an embedded EXPLAIN statement executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
COLLID	VARCHAR(128) NOT NULL	The collection ID: **DSNDYNAMICSQLCACHE** The row originates from the dynamic statement cache **DSNEXPLAINMODEYES** The row originates from an application that specifies YES for the value of the CURRENT EXPLAIN MODE special register. **DSNEXPLAINMODEEXPLAIN** The row originates from an application that specifies EXPLAIN for the value of the CURRENT EXPLAIN MODE special register. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
GROUP_MEMBER	VARCHAR(24) NOT NULL	The member name of the DB2 that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.
SECTNOI	INTEGER NOT NULL WITH DEFAULT	The section number of the statement. The value is taken from the same column in SYSPACKSTMT or SYSSTMT tables and can be used to join tables to reconstruct the access path for the statement. This column is applicable only for static statements. The default value of -1 indicates EXPLAIN information that was captured in Version 9 or earlier.
SEQNO	INTEGER NOT NULL WITH DEFAULT	The sequence number for this row if QI_DATA exceeds the size of its column.
EXPLAIN_TIME	TIMESTAMP NOT NULL	The time when the EXPLAIN information was captured: **All cached statements** When the statement entered the cache, in the form of a full-precision timestamp value. **Non-cached static statements** When the statement was bound, in the form of a full precision timestamp value. **Non-cached dynamic statements** When EXPLAIN was executed, in the form of a value equivalent to a CHAR(16) representation of the time appended by 4 zeros.
TYPE	CHAR(8) NOT NULL WITH DEFAULT	The type of the output for this row: **A** This row is for a query that DB2 attempts to run on an accelerator server. The value in column REASON_CODE indicates the outcome.

Table 191. Descriptions of columns in DSN_QUERYINFO_TABLE (continued)

Column name	Data type	Description
QI_DATA	CLOB(2M) NOT NULL WITH DEFAULT	When TYPE='A': • For REASON_CODE values other than 0, this value is the description of the REASON_CODE value. • For a REASON_CODE value of 0, this value is the query text, after it is converted for processing by the accelerator.
SERVICE_INFO	BLOB(2M) NOT NULL WITH DEFAULT	IBM internal use only.
QB_INFO_ROWID	ROWID NOT NULL GENERATED ALWAYS	IBM internal use only.
EXPANSION_REASON	CHAR(2) NOT NULL WITH DEFAULT	This column applies to only statements that reference archive tables or temporal tables. For other statements, this column is blank. Indicates the effect of the CURRENT TEMPORAL BUSINESS_TIME special register, the CURRENT TEMPORAL SYSTEM_TIME special register, and the SYSIBMADM.GET_ARCHIVE built-in global variable. These items are controlled by the BUSTIMESENSITIVE, SYSTIMESENSITIVE, and ARCHIVESENSITIVE bind options. DB2 implicitly adds certain syntax to the query if one of the following conditions are true: • The SYSIBMADM.GET_ARCHIVE global variable is set to Y and the ARCHIVESENSITIVE bind option is set to YES • The CURRENT TEMPORAL BUSINESS_TIME special register is not null and the BUSTIMESENSITIVE bind option is set to YES • The CURRENT TEMPORAL SYSTEM_TIME special register is not null and the SYSTIMESENSITIVE bind option is set to YES This column can have one of the following values: **A** The query contains implicit query transformation as a result of the SYSIBMADM.GET_ARCHIVE built-in global variable. **B** The query contains implicit query transformation as a result of the CURRENT TEMPORAL BUSINESS_TIME special register. **S** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register. **SB** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register and the CURRENT TEMPORAL BUSINESS_TIME special register. **blank** The query does not contain implicit query transformation.

Notes:

1. The REASON_CODE column has the following values:

   0 The query block qualifies for routing to an accelerator server. The values of QINAME1 and QINAME2 identify the accelerator server.

For example, for version 1 of the IBM DB2 Analytics Accelerator for z/OS, the associated data mart name is recorded in the QINAME2 column, with the following naming convention: *data-mart-name@accelerator-name@digits*.

1 No active accelerator server was found when EXPLAIN was executed.

2

Special register CURRENT QUERY ACCELERATION is set to NONE.

3 DB2 classified the query as a short-running query, or DB2 determined that sending the query to an accelerator server provided no performance advantage.

4 The query is not read-only.

6 The cursor is defined as a scrollable cursor or rowset cursor.

7 The query references objects with multiple encoding schemes.

8 The FROM clause of the query specifies a data change table reference.

9 The query contains a table expression with one or more correlated references to other tables in the same FROM clause.

10 The query contains a reference to a recursive common table expression.

11 The query contains an unsupported expression. The text of the expression is in QI_DATA.

12 The query references a table that meets one of the following conditions:

- The table is not defined in the accelerator server.
- The table is defined in the accelerator server, but is not enabled for processing by an accelerator.

13 The accelerator server that contains the tables that are referenced by the query is not started.

14 A column that is referenced in the query was altered by DB2 after the data was loaded in the accelerator server.

15 The query uses functionality that is available only in DB2 for z/OS Version 10 new-function mode or later, and the functionality is not supported by the accelerator server.

17 The query is an INSERT statement, but the DB2 subsystem parameter DSN6SPRM.QUERY_ACCEL_OPTIONS does not specify option 2 to enable its acceleration.

19 The accelerator server is not at the correct level and does not support a function in the SQL statement. The QI_DATA column contains the function text or expression text that is using the unsupported function for the accelerator server.

900-999

For IBM internal use only.

PSPI

Related reference:

Information about one example of an IBM accelerator product

DSN_QUERY_TABLE

The query table, DSN_QUERY_TABLE, contains information about a SQL statement, and displays the statement before and after query transformation.

PSPI

Unlike other EXPLAIN tables, rows in DSN_QUERY_TABLE are not populated for static SQL statements at BIND or REBIND with the EXPLAIN(YES) option.

Recommendation: Do not manually insert data into system-maintained EXPLAIN tables, and use care when deleting obsolete EXPLAIN table data. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools. Certain optimization tools depend on instances of the various EXPLAIN tables. Be careful not to delete data from or drop instances EXPLAIN tables that are created for these tools.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM
One instance of each EXPLAIN table can be created with the SYSIBM qualifier. DB2 and SQL optimization tools might use these tables. These tables are created when you run job DSNTIJSG when you install or migrate DB2.

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind. They are also populated when you specify EXPLAIN(YES) or EXPLAIN(ONLY) in a BIND or REBIND command. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions

The following table describes the columns of DSN_QUERY_TABLE.

Table 192. DSN_QUERY_TABLE description

Column name	Data type	Description
QUERYNO	INTEGER NOT NULL	A number that identifies the statement that is being explained. The origin of the value depends on the context of the row: **For rows produced by EXPLAIN statements** The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax. **For rows not produced by EXPLAIN statements** DB2 assigns a number that is based on the line number of the SQL statement in the source program. When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in certain rare cases, the value is not guaranteed to be unique. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, if the QUERYNO clause is specified, then its value is used by DB2. Otherwise DB2 assigns a number based on the line number of the SQL statement in the non-inline SQL function, native SQL procedure.
TYPE	CHAR(8) NOT NULL	The type of the data in the NODE_DATA column.
QUERY STAGE	CHAR(8) NOT NULL WITH DEFAULT	The stage during query transformation when this row is populated.
SEQNO	NOT NULL	The sequence number for this row if NODE_DATA exceeds the size of its column.
NODE_DATA	CLOB(2M)	The XML data containing the SQL statement and its query block, table, and column information.
EXPLAIN_TIME	TIMESTAMP	The EXPLAIN timestamp.
QUERY_ROWID	ROWID NOT NULL GENERATED ALWAYS	The ROWID of the statement.
GROUP MEMBER	VARCHAR(24) NOT NULL	The member name of the DB2 subsystem that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.
HASHKEY	INTEGER NOT NULL	The hash value of the contents in NODE_DATA
HAS_PRED	CHAR(1) NOT NULL	When NODE_DATA contains an SQL statement, this column indicates if the statement contains a parameter marker literal, non-parameter marker literal, or no predicates.
SECTNOI	INTEGER NOT NULL WITH DEFAULT	The section number of the statement. The value is taken from the same column in SYSPACKSTMT or SYSSTMT tables and can be used to join tables to reconstruct the access path for the statement. This column is applicable only for static statements. The default value of -1 indicates EXPLAIN information that was captured in Version 9 or earlier.

Table 192. DSN_QUERY_TABLE description (continued)

Column name	Data type	Description
APPLNAME	VARCHAR(24) NOT NULL WITH DEFAULT	The name of the application plan for the row. Applies only to embedded EXPLAIN statements that are executed from a plan or to statements that are explained when binding a plan. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
PROGNAME	VARCHAR(128) NOT NULL WITH DEFAULT	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
COLLID	VARCHAR(128) NOT NULL WITH DEFAULT	The collection ID: **DSNDYNAMICSQLCACHE** The row originates from the dynamic statement cache **DSNEXPLAINMODEYES** The row originates from an application that specifies YES for the value of the CURRENT EXPLAIN MODE special register. **DSNEXPLAINMODEEXPLAIN** The row originates from an application that specifies EXPLAIN for the value of the CURRENT EXPLAIN MODE special register. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
VERSION	VARCHAR(122) NOT NULL WITH DEFAULT	The version identifier for the package. Applies only to an embedded EXPLAIN statement executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.

Table 192. DSN_QUERY_TABLE description (continued)

Column name	Data type	Description
EXPANSION_REASON	CHAR(2) NOT NULL WITH DEFAULT	This column applies to only statements that reference archive tables or temporal tables. For other statements, this column is blank. Indicates the effect of the CURRENT TEMPORAL BUSINESS_TIME special register, the CURRENT TEMPORAL SYSTEM_TIME special register, and the SYSIBMADM.GET_ARCHIVE built-in global variable. These items are controlled by the BUSTIMESENSITIVE, SYSTIMESENSITIVE, and ARCHIVESENSITIVE bind options. DB2 implicitly adds certain syntax to the query if one of the following conditions are true: • The SYSIBMADM.GET_ARCHIVE global variable is set to Y and the ARCHIVESENSITIVE bind option is set to YES • The CURRENT TEMPORAL BUSINESS_TIME special register is not null and the BUSTIMESENSITIVE bind option is set to YES • The CURRENT TEMPORAL SYSTEM_TIME special register is not null and the SYSTIMESENSITIVE bind option is set to YES This column can have one of the following values: **A** The query contains implicit query transformation as a result of the SYSIBMADM.GET_ARCHIVE built-in global variable. **B** The query contains implicit query transformation as a result of the CURRENT TEMPORAL BUSINESS_TIME special register. **S** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register. **SB** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register and the CURRENT TEMPORAL BUSINESS_TIME special register. **blank** The query does not contain implicit query transformation.

PSPI

DSN_SORTKEY_TABLE

The sort key table, DSN_SORTKEY_TABLE, contains information about sort keys for all of the sorts required by a query.

Recommendation: Do not manually insert data into system-maintained EXPLAIN tables, and use care when deleting obsolete EXPLAIN table data. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools. Certain optimization tools depend on instances of the various EXPLAIN tables. Be careful not to delete data from or drop instances EXPLAIN tables that are created for these tools.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM
One instance of each EXPLAIN table can be created with the SYSIBM qualifier. DB2 and SQL optimization tools might use these tables. These tables are created when you run job DSNTIJSG when you install or migrate DB2.

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind. They are also populated when you specify EXPLAIN(YES) or EXPLAIN(ONLY) in a BIND or REBIND command. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions

The following table describes the columns of DSN_SORTKEY_TABLE.

Table 193. DSN_SORTKEY_TABLE description

Column name	Data type	Description
QUERYNO	INTEGER NOT NULL	A number that identifies the statement that is being explained. The origin of the value depends on the context of the row: **For rows produced by EXPLAIN statements** The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax. **For rows not produced by EXPLAIN statements** DB2 assigns a number that is based on the line number of the SQL statement in the source program. When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in certain rare cases, the value is not guaranteed to be unique. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, if the QUERYNO clause is specified, then its value is used by DB2. Otherwise DB2 assigns a number based on the line number of the SQL statement in the non-inline SQL function, native SQL procedure.
QBLOCKNO	SMALLINT NOT NULL	A number that identifies each query block within a query. The value of the numbers are not in any particular order, nor are they necessarily consecutive.
PLANNO	SMALLINT NOT NULL	The plan number, a number used to identify each miniplan with a query block.
APPLNAME	VARCHAR(24) NOT NULL	The name of the application plan for the row. Applies only to embedded EXPLAIN statements that are executed from a plan or to statements that are explained when binding a plan. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
PROGNAME	VARCHAR(128) NOT NULL	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.

Table 193. DSN_SORTKEY_TABLE description (continued)

Column name	Data type	Description
COLLID	VARCHAR(128) NOT NULL WITH DEFAULT	The collection ID: **DSNDYNAMICSQLCACHE** The row originates from the dynamic statement cache **DSNEXPLAINMODEYES** The row originates from an application that specifies YES for the value of the CURRENT EXPLAIN MODE special register. **DSNEXPLAINMODEEXPLAIN** The row originates from an application that specifies EXPLAIN for the value of the CURRENT EXPLAIN MODE special register. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
SORTNO	SMALLINT NOT NULL	The sequence number of the sort
ORDERNO	SMALLINT NOT NULL	The sequence number of the sort key
EXPTYPE	CHAR(3) NOT NULL	The type of the sort key. The possible values are: • 'COL' • 'EXP' • 'QRY'
TEXT	VARCHAR(128) NOT NULL	The sort key text, can be a column name, an expression, or a scalar subquery, or 'Record ID'.
TABNO	SMALLINT NOT NULL	The table number, a number which uniquely identifies the corresponding table reference within a query.
COLNO	SMALLINT NOT NULL	The column number, a number which uniquely identifies the corresponding column within a query. Only applicable when the sort key is a column.
DATATYPE	CHAR(18)	The data type of sort key. The possible values are • 'HEXADECIMAL' • 'CHARACTER' • 'PACKED FIELD ' • 'FIXED(31)' • 'FIXED(15)' • 'DATE' • 'TIME' • 'VARCHAR' • 'PACKED FLD' • 'FLOAT' • 'TIMESTAMP' • 'UNKNOWN DATA TYPE'
LENGTH	INTEGER NOT NULL	The length of sort key.
CCSID	INTEGER NOT NULL	IBM internal use only.

Table 193. DSN_SORTKEY_TABLE description (continued)

Column name	Data type	Description
ORDERCLASS	INTEGER NOT NULL	IBM internal use only.
EXPLAIN_TIME	TIMESTAMP NOT NULL	The time when the EXPLAIN information was captured: **All cached statements** When the statement entered the cache, in the form of a full-precision timestamp value. **Non-cached static statements** When the statement was bound, in the form of a full precision timestamp value. **Non-cached dynamic statements** When EXPLAIN was executed, in the form of a value equivalent to a CHAR(16) representation of the time appended by 4 zeros.
GROUP_MEMBER	VARCHAR(24) NOT NULL	The member name of the DB2 that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.
SECTNOI	INTEGER NOT NULL WITH DEFAULT	The section number of the statement. The value is taken from the same column in SYSPACKSTMT or SYSSTMT tables and can be used to join tables to reconstruct the access path for the statement. This column is applicable only for static statements. The default value of -1 indicates EXPLAIN information that was captured in Version 9 or earlier.
VERSION	VARCHAR(122) NOT NULL WITH DEFAULT	The version identifier for the package. Applies only to an embedded EXPLAIN statement executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.

Table 193. DSN_SORTKEY_TABLE description (continued)

Column name	Data type	Description
EXPANSION_REASON	CHAR(2) NOT NULL WITH DEFAULT	This column applies to only statements that reference archive tables or temporal tables. For other statements, this column is blank. Indicates the effect of the CURRENT TEMPORAL BUSINESS_TIME special register, the CURRENT TEMPORAL SYSTEM_TIME special register, and the SYSIBMADM.GET_ARCHIVE built-in global variable. These items are controlled by the BUSTIMESENSITIVE, SYSTIMESENSITIVE, and ARCHIVESENSITIVE bind options. DB2 implicitly adds certain syntax to the query if one of the following conditions are true: • The SYSIBMADM.GET_ARCHIVE global variable is set to Y and the ARCHIVESENSITIVE bind option is set to YES • The CURRENT TEMPORAL BUSINESS_TIME special register is not null and the BUSTIMESENSITIVE bind option is set to YES • The CURRENT TEMPORAL SYSTEM_TIME special register is not null and the SYSTIMESENSITIVE bind option is set to YES This column can have one of the following values: **A** The query contains implicit query transformation as a result of the SYSIBMADM.GET_ARCHIVE built-in global variable. **B** The query contains implicit query transformation as a result of the CURRENT TEMPORAL BUSINESS_TIME special register. **S** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register. **SB** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register and the CURRENT TEMPORAL BUSINESS_TIME special register. **blank** The query does not contain implicit query transformation.

PSPI

DSN_SORT_TABLE

The sort table, DSN_SORT_TABLE, contains information about the sort operations required by a query.

Recommendation: Do not manually insert data into system-maintained EXPLAIN tables, and use care when deleting obsolete EXPLAIN table data. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools. Certain optimization tools depend on instances of the various EXPLAIN tables. Be careful not to delete data from or drop instances EXPLAIN tables that are created for these tools.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM
One instance of each EXPLAIN table can be created with the SYSIBM qualifier. DB2 and SQL optimization tools might use these tables. These tables are created when you run job DSNTIJSG when you install or migrate DB2.

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind. They are also populated when you specify EXPLAIN(YES) or EXPLAIN(ONLY) in a BIND or REBIND command. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions

The following table describes the columns of DSN_SORT_TABLE.

Table 194. DSN_SORT_TABLE description

Column name	Data type	Description
QUERYNO	INTEGER NOT NULL	A number that identifies the statement that is being explained. The origin of the value depends on the context of the row: **For rows produced by EXPLAIN statements** The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax. **For rows not produced by EXPLAIN statements** DB2 assigns a number that is based on the line number of the SQL statement in the source program. When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in certain rare cases, the value is not guaranteed to be unique. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, if the QUERYNO clause is specified, then its value is used by DB2. Otherwise DB2 assigns a number based on the line number of the SQL statement in the non-inline SQL function, native SQL procedure.
QBLOCKNO	SMALLINT NOT NULL	A number that identifies each query block within a query. The value of the numbers are not in any particular order, nor are they necessarily consecutive.
PLANNO	SMALLINT NOT NULL	The plan number, a number used to identify each miniplan with a query block.
APPLNAME	VARCHAR(24) NOT NULL	The name of the application plan for the row. Applies only to embedded EXPLAIN statements that are executed from a plan or to statements that are explained when binding a plan. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
PROGNAME	VARCHAR(128) NOT NULL	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.

Table 194. DSN_SORT_TABLE description (continued)

Column name	Data type	Description
COLLID	VARCHAR(128) NOT NULL	The collection ID: **DSNDYNAMICSQLCACHE** The row originates from the dynamic statement cache **DSNEXPLAINMODEYES** The row originates from an application that specifies YES for the value of the CURRENT EXPLAIN MODE special register. **DSNEXPLAINMODEEXPLAIN** The row originates from an application that specifies EXPLAIN for the value of the CURRENT EXPLAIN MODE special register. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
SORTC	CHAR(5) NOT NULL WITH DEFAULT	Indicates the reasons for sort of the composite table. The reasons are shown as a series of bytes: • Byte 1 is 'G' if the reason is GROUP BY, or otherwise blank. • The second byte is 'J' if the reason is JOIN, or otherwise blank. • Byte is 'O' if the reason is ORDER BY, or otherwise blank. • The fourth by is 'U' if the reason is uniqueness, or otherwise blank.
SORTN	CHAR(5) NOT NULL WITH DEFAULT	Indicates the reasons for sort of the new table. The reasons are shown as a series of bytes: • The first byte is 'G' if the reason is GROUP BY, or otherwise blank. • The second byte is 'J' if the reason is JOIN, or otherwise blank. • The third byte is 'O' if the reason is ORDER BY, or otherwise blank. • The fourth by is 'U' if the reason is uniqueness, or otherwise blank.
SORTNO	SMALLINT NOT NULL	The sequence number of the sort.
KEYSIZE	SMALLINT NOT NULL	The sum of the lengths of the sort keys.
ORDERCLASS	INTEGER NOT NULL	IBM internal use only.
EXPLAIN_TIME	TIMESTAMP NOT NULL	The time when the EXPLAIN information was captured: **All cached statements** When the statement entered the cache, in the form of a full-precision timestamp value. **Non-cached static statements** When the statement was bound, in the form of a full precision timestamp value. **Non-cached dynamic statements** When EXPLAIN was executed, in the form of a value equivalent to a CHAR(16) representation of the time appended by 4 zeros.

Table 194. DSN_SORT_TABLE description (continued)

Column name	Data type	Description
GROUP_MEMBER	VARCHAR(24) NOT NULL	The member name of the DB2 that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.
SECTNOI	INTEGER NOT NULL WITH DEFAULT	The section number of the statement. The value is taken from the same column in SYSPACKSTMT or SYSSTMT tables and can be used to join tables to reconstruct the access path for the statement. This column is applicable only for static statements. The default value of -1 indicates EXPLAIN information that was captured in Version 9 or earlier.
VERSION	VARCHAR(122) NOT NULL WITH DEFAULT	The version identifier for the package. Applies only to an embedded EXPLAIN statement executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
EXPANSION_REASON	CHAR(2) NOT NULL WITH DEFAULT	This column applies to only statements that reference archive tables or temporal tables. For other statements, this column is blank. Indicates the effect of the CURRENT TEMPORAL BUSINESS_TIME special register, the CURRENT TEMPORAL SYSTEM_TIME special register, and the SYSIBMADM.GET_ARCHIVE built-in global variable. These items are controlled by the BUSTIMESENSITIVE, SYSTIMESENSITIVE, and ARCHIVESENSITIVE bind options. DB2 implicitly adds certain syntax to the query if one of the following conditions are true: • The SYSIBMADM.GET_ARCHIVE global variable is set to Y and the ARCHIVESENSITIVE bind option is set to YES • The CURRENT TEMPORAL BUSINESS_TIME special register is not null and the BUSTIMESENSITIVE bind option is set to YES • The CURRENT TEMPORAL SYSTEM_TIME special register is not null and the SYSTIMESENSITIVE bind option is set to YES This column can have one of the following values: **A** The query contains implicit query transformation as a result of the SYSIBMADM.GET_ARCHIVE built-in global variable. **B** The query contains implicit query transformation as a result of the CURRENT TEMPORAL BUSINESS_TIME special register. **S** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register. **SB** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register and the CURRENT TEMPORAL BUSINESS_TIME special register. **blank** The query does not contain implicit query transformation.

PSPI

DSN_STATEMENT_CACHE_TABLE

The statement cache table, DSN_STATEMENT_CACHE_TABLE, contains information about the SQL statements in the statement cache, information captured as the results of an EXPLAIN STATEMENT CACHE ALL statement.

PSPI

Recommendation: Do not manually insert data into system-maintained EXPLAIN tables, and use care when deleting obsolete EXPLAIN table data. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools. Certain optimization tools depend on instances of the various EXPLAIN tables. Be careful not to delete data from or drop instances EXPLAIN tables that are created for these tools.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM
One instance of each EXPLAIN table can be created with the SYSIBM qualifier. DB2 and SQL optimization tools might use these tables. These tables are created when you run job DSNTIJSG when you install or migrate DB2.

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind. They are also populated when you specify EXPLAIN(YES) or EXPLAIN(ONLY) in a BIND or REBIND command. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions

The following table shows the descriptions of the columns in DSN_STATEMENT_CACHE_TABLE.

Table 195. Descriptions of columns in DSN_STATEMENT_CACHE_TABLE

Column name	Data Type	Description
STMT_ID	INTEGER NOT NULL	The statement ID; this value is the EDM unique token for the statement.
STMT_TOKEN	VARCHAR(240)	The statement token; you provide this value as an identification string.

Table 195. Descriptions of columns in DSN_STATEMENT_CACHE_TABLE (continued)

Column name	Data Type	Description
COLLID	VARCHAR(128) NOT NULL	The collection ID: **DSNDYNAMICSQLCACHE** The row originates from the dynamic statement cache **DSNEXPLAINMODEYES** The row originates from an application that specifies YES for the value of the CURRENT EXPLAIN MODE special register. **DSNEXPLAINMODEEXPLAIN** The row originates from an application that specifies EXPLAIN for the value of the CURRENT EXPLAIN MODE special register.
PROGRAM_NAME	VARCHAR(128) NOT NULL	The name of the package that performed the initial PREPARE for the statement.
INV_DROPALT	CHAR(1) NOT NULL	This column indicates if the statement has been invalidated by a DROP or ALTER statement.3 on page 2572
INV_REVOKE	CHAR(1) NOT NULL	This column indicates if the statement has been invalidated by a REVOKE statement.3 on page 2572
INV_LRU	CHAR(1) NOT NULL	This column indicates if the statement has been removed from the cache by LRU.3 on page 2572
INV_RUNSTATS	CHAR(1) NOT NULL	This column indicates if the statement has been invalidated by RUNSTATS.3 on page 2572
CACHED_TS	TIMESTAMP NOT NULL	The timestamp when the statement was stored in the dynamic statement cache.3 on page 2572
USERS	INTEGER NOT NULL	The number of current users of the statement. This number indicates the users that have prepared or run the statement during their current unit of work. 1 on page 2572,3 on page 2572
COPIES	INTEGER NOT NULL	The number of copies of the statement that are owned by all threads in the system. 1 on page 2572,3 on page 2572
LINES	INTEGER NOT NULL	The precompiler line number from the initial PREPARE of the statement. 1 on page 2572
PRIMAUTH	VARCHAR(128) NOT NULL	The primary authorization ID that did the initial PREPARE of the statement.
CURSQLID	VARCHAR(128) NOT NULL	The CURRENT SQLID that did the initial PREPARE of the statement.
BIND_QUALIFIER	VARCHAR(128) NOT NULL	The BIND qualifier. For unqualified table names, this is the object qualifier.
BIND_ISO	CHAR(2) NOT NULL	The value of the ISOLATION BIND option that is in effect for this statement. The value will be one of the following values: **'UR'** Uncommitted read **'CS'** Cursor stability **'RS'** Read stability **'RR'** Repeatable read
BIND_CDATA	CHAR(1) NOT NULL	The value of the CURRENTDATA BIND option that is in effect for this statement. The value will be one of the following values: **'Y'** CURRENTDATA(YES) **'N'** CURRENTDATA(NO)

Table 195. Descriptions of columns in DSN_STATEMENT_CACHE_TABLE (continued)

Column name	Data Type	Description
BIND_DYNRL	CHAR(1) NOT NULL	The value of the DYNAMICRULES BIND option that is in effect for this statement. The value will be one of the following values: **'B'** DYNAMICRULE(BIND) **'R'** DYNAMICRULES(RUN)
BIND_DEGRE	CHAR(1) NOT NULL	The value of the CURRENT DEGREE special register that is in effect for this statement. The value will be one of the following values: **'A'** CURRENT DEGREE = ANY **'1'** CURRENT DEGREE = 1
BIND_SQLRL	CHAR(1) NOT NULL	The value of the CURRENT RULES special register that is in effect for this statement. The value will be one of the following values: **'D'** CURRENT RULES = DB2 **'S'** CURRENT RULES = SQL
BIND_CHOLD	CHAR(1) NOT NULL	The value of the WITH HOLD attribute of the PREPARE for this statement. The value will be one of the following values: **'Y'** Initial PREPARE specified WITH HOLD **'N'** Initial PREPARE specified WITHOUT HOLD
STAT_TS	TIMESTAMP NOT NULL	Timestamp of the statistics. This is the timestamp when IFCID 318 is started. 2 on page 2572
STAT_EXEC	INTEGER NOT NULL	This column is deprecated. Use STAT_EXECB instead.
STAT_GPAG	INTEGER NOT NULL	This column is deprecated. Use STAT_GPAGB instead. 1 on page 2572
STAT_SYNR	INTEGER NOT NULL	This column is deprecated. Use STAT_SYNRB instead. 1 on page 2572
STAT_WRIT	INTEGER NOT NULL	This column is deprecated. Use STAT_WRITB instead. 1 on page 2572
STAT_EROW	INTEGER NOT NULL	This column is deprecated. Use STAT_EROWB instead. 1 on page 2572
STAT_PROW	INTEGER NOT NULL	This column is deprecated. Use STAT_PROWB instead. 1 on page 2572
STAT_SORT	INTEGER NOT NULL	This column is deprecated. Use STAT_SORTB instead. 1 on page 2572
STAT_INDX	INTEGER NOT NULL	This column is deprecated. Use STAT_SORTB instead.
STAT_RSCN	INTEGER NOT NULL	This column is deprecated. Use STAT_SORTB instead.
STAT_PGRP	INTEGER NOT NULL	This column is deprecated. Use STAT_SORTB instead.
STAT_ELAP	FLOAT NOT NULL	The accumulated elapsed time that is used for the statement. 2 on page 2572
STAT_CPU	FLOAT NOT NULL	The accumulated CPU time that is used for the statement. 2 on page 2572
STAT_SUS_SYNIO	FLOAT NOT NULL	The accumulated wait time for synchronous I/O operations for the statement. 2 on page 2572
STAT_SUS_LOCK	FLOAT NOT NULL	The accumulated wait time for lock requests for the statement. 2 on page 2572
STAT_SUS_SWIT	FLOAT NOT NULL	The accumulated wait time for synchronous execution unit switch for the statement. 2 on page 2572
STAT_SUS_GLCK	FLOAT NOT NULL	The accumulated wait time for global locks for this statement. 2 on page 2572

Table 195. Descriptions of columns in DSN_STATEMENT_CACHE_TABLE (continued)

Column name	Data Type	Description
STAT_SUS_OTHR	FLOAT NOT NULL	The accumulated wait time for read activity that is done by another thread. 2 on page 2572
STAT_SUS_OTHW	FLOAT NOT NULL	The accumulated wait time for write activity done by another thread. 2 on page 2572
STAT_RIDLIMT	INTEGER NOT NULL	This column is deprecated. Use STAT_SORTB instead.
STAT_RIDSTOR	INTEGER NOT NULL	This column is deprecated. Use STAT_SORTB instead.
EXPLAIN_TS	TIMESTAMP NOT NULL	The timestamp for when the statement cache table is populated.
SCHEMA	VARCHAR(128) NOT NULL	The value of the CURRENT SCHEMA special register.
STMT_TEXT	CLOB(2M) NOT NULL	The statement that is being explained.
STMT_ROWID	ROWID NOT NULL GENERATED ALWAYS	The ROWID of the statement.
BIND_RO_TYPE	CHAR(1) NOT NULL WITH DEFAULT	The current specification of the REOPT option for the statement3 on page 2572: **'N'** REOPT(NONE) or its equivalent **'1'** REOPT(ONCE) or its equivalent **'A'** REOPT(AUTO) or its equivalent **'O'** The current plan is deemed optimal and there is no need for REOPT(AUTO)
BIND_RA_TOT	INTEGER NOT NULL WITH DEFAULT	The total number of REBIND commands that have been issued for the dynamic statement because of the REOPT(AUTO) option.1 on page 2572,3 on page 2572
GROUP_MEMBER	VARCHAR(24) NOT NULL WITH DEFAULT	The member name of the DB2 that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.
STAT_EXECB	BIGINT NOT NULL WITH DEFAULT	The number of times this statement has been run. For a statement with a cursor, this is the number of OPENs.2 on page 2572
STAT_GPAGB	BIGINT NOT NULL WITH DEFAULT	The number of getpage operations that are performed for the statement. 2 on page 2572
STAT_SYNRB	BIGINT NOT NULL WITH DEFAULT	The number of synchronous buffer reads that are performed for the statement. 2 on page 2572
STAT_WRITB	BIGINT NOT NULL WITH DEFAULT	The number of buffer write operations that are performed for the statement. 2 on page 2572
STAT_EROWB	BIGINT NOT NULL WITH DEFAULT	The number of rows that are examined for the statement. 2 on page 2572
STAT_PROWB	BIGINT NOT NULL WITH DEFAULT	The number of rows that are processed for the statement. 2 on page 2572
STAT_SORTB	BIGINT NOT NULL WITH DEFAULT	The number of sorts that are performed for the statement.2 on page 2572
STAT_INDXB	BIGINT NOT NULL WITH DEFAULT	The number of index scans that are performed for the statement.2 on page 2572
STAT_RSCNB	BIGINT NOT NULL WITH DEFAULT	The number of table space scans that are performed for the statement.2 on page 2572
STAT_PGRPB	BIGINT NOT NULL WITH DEFAULT	The number of parallel groups that are created for the statement.2 on page 2572

Table 195. Descriptions of columns in DSN_STATEMENT_CACHE_TABLE (continued)

Column name	Data Type	Description
STAT_RIDLIMTB	BIGINT NOT NULL WITH DEFAULT	The number of times a RID list was not used because the number of RIDs would have exceeded DB2 limits.2 on page 2572
STAT_RIDSTORB	BIGINT NOT NULL WITH DEFAULT	.The number of times a RID list was not used because there is not enough storage available to hold the list of RIDs.2 on page 2572
LITERAL_REPL	CHAR(1) NOT NULL WITH DEFAULT	Identifies cached statements where the literal values are replaced by the '&' symbol:3 on page 2572 **'R'** The statement is prepared with CONCENTRATE STATEMENTS WITH LITERALS behavior and the literal constants in the statement have been replaced with '&' . **'D'** This statement is a duplicate statement instance with different literal reusability criteria. **blank** Literal values are not replaced.
STAT_SUS_LATCH	FLOAT NOT NULL WITH DEFAULT	The accumulated wait time for latch requests for the statement.
STAT_SUS_PLATCH	FLOAT NOT NULL WITH DEFAULT	The accumulated wait time for page latch requests for the statement.
STAT_SUS_DRAIN	FLOAT NOT NULL WITH DEFAULT	The accumulated wait time for drain lock requests for the statement.
STAT_SUS_CLAIM	FLOAT NOT NULL WITH DEFAULT	The accumulated wait time for claim count requests for the statement.
STAT_SUS_LOG	FLOAT NOT NULL WITH DEFAULT	The accumulated wait time for log writer requests for the statement.

Table 195. Descriptions of columns in DSN_STATEMENT_CACHE_TABLE (continued)

Column name	Data Type	Description
EXPANSION_REASON	CHAR(2) NOT NULL WITH DEFAULT	This column applies to only statements that reference archive tables or temporal tables. For other statements, this column is blank. Indicates the effect of the CURRENT TEMPORAL BUSINESS_TIME special register, the CURRENT TEMPORAL SYSTEM_TIME special register, and the SYSIBMADM.GET_ARCHIVE built-in global variable. These items are controlled by the BUSTIMESENSITIVE, SYSTIMESENSITIVE, and ARCHIVESENSITIVE bind options. DB2 implicitly adds certain syntax to the query if one of the following conditions are true: • The SYSIBMADM.GET_ARCHIVE global variable is set to Y and the ARCHIVESENSITIVE bind option is set to YES • The CURRENT TEMPORAL BUSINESS_TIME special register is not null and the BUSTIMESENSITIVE bind option is set to YES • The CURRENT TEMPORAL SYSTEM_TIME special register is not null and the SYSTIMESENSITIVE bind option is set to YES This column can have one of the following values: **A** The query contains implicit query transformation as a result of the SYSIBMADM.GET_ARCHIVE built-in global variable. **B** The query contains implicit query transformation as a result of the CURRENT TEMPORAL BUSINESS_TIME special register. **S** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register. **SB** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register and the CURRENT TEMPORAL BUSINESS_TIME special register. **blank** The query does not contain implicit query transformation.

Notes:

1. If the specified value exceeds 2147483647, the column contains the value 2147483647.
2. Statistics are cumulative, across executions of the same statement, and across threads, if the value of COLLID is DSNDYNAMICSQLCACHE. If the value of COLLID is DSNEXPLAINMODEYES, the values are for a single run of the statement only. If the value of COLLID is DSNEXPLAINMODE EXPLAIN, the values of all statistics columns are 0.
3. The column is not applicable when the value of the COLLID column is 'DSNEXPLAINMODEYES' or 'DSNEXPLAINMODEEXPLAIN'

PSPI

DSN_STATEMNT_TABLE

The statement table, DSN_STATEMNT_TABLE, contains information about the estimated cost of specified SQL statements.

Recommendation: Do not manually insert data into system-maintained EXPLAIN tables, and use care when deleting obsolete EXPLAIN table data. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools. Certain optimization tools depend on instances of the various EXPLAIN tables. Be careful not to delete data from or drop instances EXPLAIN tables that are created for these tools.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM
One instance of each EXPLAIN table can be created with the SYSIBM qualifier. DB2 and SQL optimization tools might use these tables. These tables are created when you run job DSNTIJSG when you install or migrate DB2.

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind. They are also populated when you specify EXPLAIN(YES) or EXPLAIN(ONLY) in a BIND or REBIND command. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions

The following table describes the content of each column in STATEMNT_TABLE.

Table 196. Descriptions of columns in DSN_STATEMNT_TABLE

Column name	Data type	Description
QUERYNO	INTEGER NOT NULL WITH DEFAULT	A number that identifies the statement that is being explained. The origin of the value depends on the context of the row: **For rows produced by EXPLAIN statements** The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax. **For rows not produced by EXPLAIN statements** DB2 assigns a number that is based on the line number of the SQL statement in the source program. When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in certain rare cases, the value is not guaranteed to be unique. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, if the QUERYNO clause is specified, then its value is used by DB2. Otherwise DB2 assigns a number based on the line number of the SQL statement in the non-inline SQL function, native SQL procedure.
APPLNAME	VARCHAR(24) NOT NULL WITH DEFAULT	The name of the application plan for the row. Applies only to embedded EXPLAIN statements that are executed from a plan or to statements that are explained when binding a plan. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
PROGNAME	VARCHAR(128) NOT NULL WITH DEFAULT	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
COLLID	VARCHAR(128) NOT NULL WITH DEFAULT	The collection ID: **DSNDYNAMICSQLCACHE** The row originates from the dynamic statement cache **DSNEXPLAINMODEYES** The row originates from an application that specifies YES for the value of the CURRENT EXPLAIN MODE special register. **DSNEXPLAINMODEEXPLAIN** The row originates from an application that specifies EXPLAIN for the value of the CURRENT EXPLAIN MODE special register. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.

Table 196. Descriptions of columns in DSN_STATEMNT_TABLE (continued)

Column name	Data type	Description
GROUP_MEMBER	VARCHAR(24) NOT NULL WITH DEFAULT	The member name of the DB2 that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.
EXPLAIN_TIME	TIMESTAMP NOT NULL WITH DEFAULT	The time when the EXPLAIN information was captured: **All cached statements** When the statement entered the cache, in the form of a full-precision timestamp value. **Non-cached static statements** When the statement was bound, in the form of a full precision timestamp value. **Non-cached dynamic statements** When EXPLAIN was executed, in the form of a value equivalent to a CHAR(16) representation of the time appended by 4 zeros.
STMT_TYPE	CHAR(6) NOT NULL WITH DEFAULT	The type of statement being explained. Possible values are: **SELECT** SELECT **INSERT** INSERT **UPDATE** UPDATE **MERGE** MERGE **DELETE** DELETE **TRUNCA** TRUNCATE **SELUPD** SELECT with FOR UPDATE OF **DELCUR** DELETE WHERE CURRENT OF CURSOR **UPDCUR** UPDATE WHERE CURRENT OF CURSOR
COST_CATEGORY	CHAR(1) NOT NULL WITH DEFAULT	Indicates if DB2 was forced to use default values when making its estimates. Possible values: **A** Indicates that DB2 had enough information to make a cost estimate without using default values. **B** Indicates that some condition exists for which DB2 was forced to use default values. See the values in REASON to determine why DB2 was unable to put this estimate in cost category A.
PROCMS	INTEGER NOT NULL WITH DEFAULT	The estimated processor cost, in milliseconds, for the SQL statement. The estimate is rounded up to the next integer value. The maximum value for this cost is 2147483647 milliseconds, which is equivalent to approximately 24.8 days. If the estimated value exceeds this maximum, the maximum value is reported. If an accelerator is used, the difference is reflected in this value.

Table 196. Descriptions of columns in DSN_STATEMNT_TABLE (continued)

Column name	Data type	Description
PROCSU	INTEGER NOT NULL WITH DEFAULT	The estimated processor cost, in service units, for the SQL statement. The estimate is rounded up to the next integer value. The maximum value for this cost is 2147483647 service units. If the estimated value exceeds this maximum, the maximum value is reported. If an accelerator is used, this value represents the estimated cost including any impact of acceleration.
REASON	VARCHAR(254) WITH DEFAULT	A string that indicates the reasons for putting an estimate into cost category B. **HAVING CLAUSE** A subselect in the SQL statement contains a HAVING clause. **HOST VARIABLES** The statement uses host variables, parameter markers, or special registers. **MATERIALIZATION** Statistics are missing because the statement uses materialized views or nested table expresses. **PROFILEID** *value* When profile monitoring is used for the statement, the value of the PROFILEID column in SYSIBM.DSN_PROFILE_TABLE. **REFERENTIAL CONSTRAINTS** Referential constraints of the type CASCADE or SET NULL exist on the target table of a DELETE statement. **TABLE CARDINALITY** The cardinality statistics are missing for one or more of the tables that are used in the statement. **TRIGGERS** Triggers are defined on the target table of an insert, update, or delete operation. **UDF** The statement uses user-defined functions.
STMT_ENCODE	CHAR(1) WITH DEFAULT	Encoding scheme of the statement. If the statement represents a single CCSID set, the possible values are: **A** ASCII **E** EBCDIC **U** Unicode If the statement has multiple CCSID sets, the value is M.
TOTAL_COST	FLOAT NOT NULL WITH DEFAULT	The overall estimated cost of the statement. If an accelerator is used the benefit is reflected in this value. This cost should be used only for reference purposes.
SECTNOI	INTEGER NOT NULL WITH DEFAULT	The section number of the statement. The value is taken from the same column in SYSPACKSTMT or SYSSTMT tables and can be used to join tables to reconstruct the access path for the statement. This column is applicable only for static statements. The default value of -1 indicates EXPLAIN information that was captured in Version 9 or earlier.

Table 196. Descriptions of columns in DSN_STATEMNT_TABLE (continued)

Column name	Data type	Description
VERSION	VARCHAR(122) NOT NULL WITH DEFAULT	The version identifier for the package. Applies only to an embedded EXPLAIN statement executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
EXPANSION_REASON	CHAR(2) NOT NULL WITH DEFAULT	This column applies to only statements that reference archive tables or temporal tables. For other statements, this column is blank. Indicates the effect of the CURRENT TEMPORAL BUSINESS_TIME special register, the CURRENT TEMPORAL SYSTEM_TIME special register, and the SYSIBMADM.GET_ARCHIVE built-in global variable. These items are controlled by the BUSTIMESENSITIVE, SYSTIMESENSITIVE, and ARCHIVESENSITIVE bind options. DB2 implicitly adds certain syntax to the query if one of the following conditions are true: • The SYSIBMADM.GET_ARCHIVE global variable is set to Y and the ARCHIVESENSITIVE bind option is set to YES • The CURRENT TEMPORAL BUSINESS_TIME special register is not null and the BUSTIMESENSITIVE bind option is set to YES • The CURRENT TEMPORAL SYSTEM_TIME special register is not null and the SYSTIMESENSITIVE bind option is set to YES This column can have one of the following values: **A** The query contains implicit query transformation as a result of the SYSIBMADM.GET_ARCHIVE built-in global variable. **B** The query contains implicit query transformation as a result of the CURRENT TEMPORAL BUSINESS_TIME special register. **S** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register. **SB** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register and the CURRENT TEMPORAL BUSINESS_TIME special register. **blank** The query does not contain implicit query transformation.

PSPI

Related reference:

Information about one example of an IBM accelerator product

DSN_STAT_FEEDBACK

The DSN_STAT_FEEDBACK table contains recommendations for capturing missing or conflicting statistics that are defined during EXPLAIN. Collecting these statistics by the RUNSTATS utility might improve the performance of the query.

PSPI The values in this table are updated only at EXPLAIN time, and are not modified by the RUNSTATS utility.

Recommendation: Do not manually insert data into system-maintained EXPLAIN tables, and use care when deleting obsolete EXPLAIN table data. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools. Certain optimization tools depend on instances of the various EXPLAIN tables. Be careful not to delete data from or drop instances EXPLAIN tables that are created for these tools.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM
One instance of each EXPLAIN table can be created with the SYSIBM qualifier. DB2 and SQL optimization tools might use these tables. These tables are created when you run job DSNTIJSG when you install or migrate DB2.

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind. They are also populated when you specify EXPLAIN(YES) or EXPLAIN(ONLY) in a BIND or REBIND command. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions

The following table contains descriptions of the columns in the DSN_STAT_FEEDBACK table.

Table 197. Descriptions of columns in the DSN_STAT_FEEDBACK table

Column name	Data Type	Descriptions
QUERYNO	INTEGER NOT NULL	A number that identifies the statement that is being explained. The origin of the value depends on the context of the row: **For rows produced by EXPLAIN statements** The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax. **For rows not produced by EXPLAIN statements** DB2 assigns a number that is based on the line number of the SQL statement in the source program. When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in certain rare cases, the value is not guaranteed to be unique. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, if the QUERYNO clause is specified, then its value is used by DB2. Otherwise DB2 assigns a number based on the line number of the SQL statement in the non-inline SQL function, native SQL procedure.
APPLNAME	VARCHAR(24) NOT NULL	The name of the application plan for the row. Applies only to embedded EXPLAIN statements that are executed from a plan or to statements that are explained when binding a plan. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
PROGNAME	VARCHAR(128) NOT NULL	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
COLLID	VARCHAR(128) NOT NULL WITH DEFAULT	The collection ID: **DSNDYNAMICSQLCACHE** The row originates from the dynamic statement cache **DSNEXPLAINMODEYES** The row originates from an application that specifies YES for the value of the CURRENT EXPLAIN MODE special register. **DSNEXPLAINMODEEXPLAIN** The row originates from an application that specifies EXPLAIN for the value of the CURRENT EXPLAIN MODE special register. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
GROUP_MEMBER	VARCHAR(24) NOT NULL	The member name of the DB2 that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.

Table 197. Descriptions of columns in the DSN_STAT_FEEDBACK table (continued)

Column name	Data Type	Descriptions
EXPLAIN_TIME	TIMESTAMP NOT NULL	The time when the EXPLAIN information was captured: **All cached statements** When the statement entered the cache, in the form of a full-precision timestamp value. **Non-cached static statements** When the statement was bound, in the form of a full precision timestamp value. **Non-cached dynamic statements** When EXPLAIN was executed, in the form of a value equivalent to a CHAR(16) representation of the time appended by 4 zeros.
SECTNOI	INTEGER NOT NULL WITH DEFAULT WITH DEFAULT	The section number of the statement. The value is taken from the same column in SYSPACKSTMT or SYSSTMT tables and can be used to join tables to reconstruct the access path for the statement. This column is applicable only for static statements. The default value of -1 indicates EXPLAIN information that was captured in Version 9 or earlier.
VERSION	VARCHAR(122) NOT NULL WITH DEFAULT	The version identifier for the package. Applies only to an embedded EXPLAIN statement executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
TBCREATOR	VARCHAR(128) NOT NULL	The creator of the table.
TBNAME	VARCHAR(128) NOT NULL	The name of the table.
IXCREATOR	VARCHAR(128) NOT NULL	The creator of the index.
IXNAME	VARCHAR(128) NOT NULL	The name of the index.
COLNAME	VARCHAR(128) NOT NULL	The name of the column.
NUMCOLUMNS	SMALLINT NOT NULL	The number of columns in the column group.
COLGROUPCOLNO	VARCHAR(254) NOT NULL FOR BIT DATA	A hex representation that identifies the set of columns associated with the statistics. If the statistics are only associated with a single column, the field contains a zero length. Otherwise, the field is an array of SMALLINT column numbers with a dimension equal to the value in NUMCOLUMNS.
TYPE	CHAR(1) NOT NULL	The type of statistic to collect: **'C'** Cardinality. **'F'** Frequency. **'H'** Histogram. **'I'** Index. **'T'** Table.

Table 197. Descriptions of columns in the DSN_STAT_FEEDBACK table (continued)

Column name	Data Type	Descriptions
DBNAME	VARCHAR(24) NOT NULL	The name of the database.
TSNAME	VARCHAR(24) NOT NULL	The name of the table space.
REASON	CHAR(8) NOT NULL	The reason that the statistic was recommend: **'BASIC'** A basic statistic value for a column table or index is missing. No statistics were collected for the identified object. **'KEYCARD'** The cardinalities of index key columns are missing. **'LOWCARD'** The cardinality of the column is a low value, which indicates that data might be skewed. **'NULLABLE'** Distribution statistics are not available for a nullable column, which indicates that data might be skewed. **'DEFAULT'** A predicate references a value that is probably a default value, which indicates that data might be skewed. **'RANGEPRD'** Histogram statistics are not available for a range predicate. **'PARALLEL'** Parallelism could be improved by uniform partitioning of key ranges. **'CONFLICT'** Another statistic contains a value that conflicts with the value of this statistic. Such conflicts usually occur because statistics were collected for related objects at different times. **'COMPFFIX'** Multi-column cardinality statistics are needed for an index compound filter factor.
REMARKS	VARCHAR(254) NOT NULL	Free form text for extensibility.

PSPI

Related tasks:

Maintaining statistics in the catalog (DB2 Performance)

Related reference:

RUNSTATS (DB2 Utilities)

Statistics used for access path selection (DB2 Performance)

"SYSIBM.SYSSTATFEEDBACK table" on page 2370

DSN_STRUCT_TABLE

The structure table, DSN_STRUCT_TABLE, contains information about all of the query blocks in a query.

PSPI

Recommendation: Do not manually insert data into system-maintained EXPLAIN tables, and use care when deleting obsolete EXPLAIN table data. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools. Certain optimization tools depend on instances of the various EXPLAIN tables. Be careful not to delete data from or drop instances EXPLAIN tables that are created for these tools.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM
One instance of each EXPLAIN table can be created with the SYSIBM qualifier. DB2 and SQL optimization tools might use these tables. These tables are created when you run job DSNTIJSG when you install or migrate DB2.

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind. They are also populated when you specify EXPLAIN(YES) or EXPLAIN(ONLY) in a BIND or REBIND command. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions

The following table describes the columns of DSN_STRUCT_TABLE

Table 198. DSN_STRUCT_TABLE description

Column name	Data type	Description
QUERYNO	INTEGER NOT NULL	A number that identifies the statement that is being explained. The origin of the value depends on the context of the row: **For rows produced by EXPLAIN statements** The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax. **For rows not produced by EXPLAIN statements** DB2 assigns a number that is based on the line number of the SQL statement in the source program. When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in certain rare cases, the value is not guaranteed to be unique. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, if the QUERYNO clause is specified, then its value is used by DB2. Otherwise DB2 assigns a number based on the line number of the SQL statement in the non-inline SQL function, native SQL procedure.
QBLOCKNO	SMALLINT NOT NULL	A number that identifies each query block within a query. The value of the numbers are not in any particular order, nor are they necessarily consecutive.
APPLNAME	VARCHAR(24) NOT NULL	The name of the application plan for the row. Applies only to embedded EXPLAIN statements that are executed from a plan or to statements that are explained when binding a plan. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
PROGNAME	VARCHAR(128) NOT NULL	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
PARENT	SMALLINT NOT NULL	The parent query block number of the current query block in the structure of SQL text; this is the same as the PARENT_QBLOCKNO in PLAN_TABLE.
TIMES	FLOAT NOT NULL	The estimated number of rows returned by Data Manager; also the estimated number of times this query block is executed.
ROWCOUNT	INTEGER NOT NULL	The estimated number of rows returned by RDS (Query Cardinality).
ATOPEN	CHAR(1) NOT NULL	Whether the query block is moved up for do-at-open processing; 'Y' if done-at-open; 'N': otherwise.

Table 198. DSN_STRUCT_TABLE description (continued)

Column name	Data type	Description
CONTEXT	CHAR(10) NOT NULL	This column indicates what the context of the current query block is. The possible values are: • 'TOP LEVEL' • 'UNION' • 'UNION ALL' • 'PREDICATE' • 'TABLE EXP' • 'UNKNOWN'
ORDERNO	SMALLINT NOT NULL	Not currently used.
DOATOPEN_PARENT	SMALLINT NOT NULL	The parent query block number of the current query block; Do-at-open parent if the query block is done-at-open, this may be different from the PARENT_QBLOCKNO in PLAN_TABLE.
QBLOCK_TYPE	CHAR(6) NOT NULL WITH DEFAULT	This column indicates the type of the current query block. The possible values are • 'SELECT' • 'INSERT' • 'UPDATE' • 'DELETE' • 'SELUPD' • 'DELCUR' • ''UPDCUR' • 'CORSUB' • 'NCOSUB' • 'TABLEX' • 'TRIGGR' • 'UNION' • 'UNIONA' • 'CTE' It is equivalent to QBLOCK_TYPE column in PLAN_TABLE, except for CTE.
EXPLAIN_TIME	TIMESTAMP NOT NULL	The time when the EXPLAIN information was captured: **All cached statements** When the statement entered the cache, in the form of a full-precision timestamp value. **Non-cached static statements** When the statement was bound, in the form of a full precision timestamp value. **Non-cached dynamic statements** When EXPLAIN was executed, in the form of a value equivalent to a CHAR(16) representation of the time appended by 4 zeros.
QUERY_STAGE	CHAR(8) NOT NULL	IBM internal use only.
GROUP_MEMBER	VARCHAR(24) NOT NULL	The member name of the DB2 that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.

Table 198. DSN_STRUCT_TABLE description (continued)

Column name	Data type	Description
ORIGIN	CHAR(1) NOT NULL WITH DEFAULT	Indicates the origin of the query block: **Blank** Generated by DB2 **C** Column mask **R** Row permission **U** Specified by the user
SECTNOI	INTEGER NOT NULL WITH DEFAULT	The section number of the statement. The value is taken from the same column in SYSPACKSTMT or SYSSTMT tables and can be used to join tables to reconstruct the access path for the statement. This column is applicable only for static statements. The default value of -1 indicates EXPLAIN information that was captured in Version 9 or earlier.
COLLID	VARCHAR(128) NOT NULL WITH DEFAULT	The collection ID: **DSNDYNAMICSQLCACHE** The row originates from the dynamic statement cache **DSNEXPLAINMODEYES** The row originates from an application that specifies YES for the value of the CURRENT EXPLAIN MODE special register. **DSNEXPLAINMODEEXPLAIN** The row originates from an application that specifies EXPLAIN for the value of the CURRENT EXPLAIN MODE special register. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
VERSION	VARCHAR(122) NOT NULL WITH DEFAULT	The version identifier for the package. Applies only to an embedded EXPLAIN statement executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.

Table 198. DSN_STRUCT_TABLE description (continued)

Column name	Data type	Description
EXPANSION_REASON	CHAR(2) NOT NULL WITH DEFAULT	This column applies to only statements that reference archive tables or temporal tables. For other statements, this column is blank. Indicates the effect of the CURRENT TEMPORAL BUSINESS_TIME special register, the CURRENT TEMPORAL SYSTEM_TIME special register, and the SYSIBMADM.GET_ARCHIVE built-in global variable. These items are controlled by the BUSTIMESENSITIVE, SYSTIMESENSITIVE, and ARCHIVESENSITIVE bind options. DB2 implicitly adds certain syntax to the query if one of the following conditions are true: • The SYSIBMADM.GET_ARCHIVE global variable is set to Y and the ARCHIVESENSITIVE bind option is set to YES • The CURRENT TEMPORAL BUSINESS_TIME special register is not null and the BUSTIMESENSITIVE bind option is set to YES • The CURRENT TEMPORAL SYSTEM_TIME special register is not null and the SYSTIMESENSITIVE bind option is set to YES This column can have one of the following values: **A** The query contains implicit query transformation as a result of the SYSIBMADM.GET_ARCHIVE built-in global variable. **B** The query contains implicit query transformation as a result of the CURRENT TEMPORAL BUSINESS_TIME special register. **S** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register. **SB** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register and the CURRENT TEMPORAL BUSINESS_TIME special register. **blank** The query does not contain implicit query transformation.

PSPI

DSN_VIEWREF_TABLE

The view reference table, DSN_VIEWREF_TABLE, contains information about all of the views and materialized query tables that are used to process a query.

PSPI

Recommendation: Do not manually insert data into system-maintained EXPLAIN tables, and use care when deleting obsolete EXPLAIN table data. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools. Certain optimization tools depend on instances of the various EXPLAIN tables. Be careful not to delete data from or drop instances EXPLAIN tables that are created for these tools.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM
One instance of each EXPLAIN table can be created with the SYSIBM qualifier. DB2 and SQL optimization tools might use these tables. These tables are created when you run job DSNTIJSG when you install or migrate DB2.

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind. They are also populated when you specify EXPLAIN(YES) or EXPLAIN(ONLY) in a BIND or REBIND command. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions

The following table describes the columns of DSN_VIEWREF_TABLE.

Table 199. DSN_VIEWREF_TABLE description

Column name	Data type	Description
QUERYNO	INTEGER NOT NULL WITH DEFAULT	A number that identifies the statement that is being explained. The origin of the value depends on the context of the row: **For rows produced by EXPLAIN statements** The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax. **For rows not produced by EXPLAIN statements** DB2 assigns a number that is based on the line number of the SQL statement in the source program. When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in certain rare cases, the value is not guaranteed to be unique. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, if the QUERYNO clause is specified, then its value is used by DB2. Otherwise DB2 assigns a number based on the line number of the SQL statement in the non-inline SQL function, native SQL procedure.
APPLNAME	VARCHAR(24) NOT NULL WITH DEFAULT	The name of the application plan for the row. Applies only to embedded EXPLAIN statements that are executed from a plan or to statements that are explained when binding a plan. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
PROGNAME	VARCHAR(128) NOT NULL WITH DEFAULT	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
VERSION	VARCHAR(122) NOT NULL WITH DEFAULT	The version identifier for the package. Applies only to an embedded EXPLAIN statement executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.

Table 199. DSN_VIEWREF_TABLE description (continued)

Column name	Data type	Description
COLLID	VARCHAR(128) NOT NULL WITH DEFAULT	The collection ID: **DSNDYNAMICSQLCACHE** The row originates from the dynamic statement cache **DSNEXPLAINMODEYES** The row originates from an application that specifies YES for the value of the CURRENT EXPLAIN MODE special register. **DSNEXPLAINMODEEXPLAIN** The row originates from an application that specifies EXPLAIN for the value of the CURRENT EXPLAIN MODE special register. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.
CREATOR	VARCHAR(128) NOT NULL WITH DEFAULT	Authorization ID of the owner of the object.
NAME	VARCHAR(128)	Name of the object.
TYPE	CHAR(1) NOT NULL WITH DEFAULT	The type of the object: **'V'** View **'R'** MQT that has been used to replace the base table for rewrite **'M'** MQT
MQTUSE	SMALLINT WITH DEFAULT	IBM internal use only.
EXPLAIN_TIME	TIMESTAMP NOT NULL WITH DEFAULT	The time when the EXPLAIN information was captured: **All cached statements** When the statement entered the cache, in the form of a full-precision timestamp value. **Non-cached static statements** When the statement was bound, in the form of a full precision timestamp value. **Non-cached dynamic statements** When EXPLAIN was executed, in the form of a value equivalent to a CHAR(16) representation of the time appended by 4 zeros.
GROUP_MEMBER	VARCHAR(24) NOT NULL	The member name of the DB2 that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.
SECTNOI	INTEGER NOT NULL WITH DEFAULT	The section number of the statement. The value is taken from the same column in SYSPACKSTMT or SYSSTMT tables and can be used to join tables to reconstruct the access path for the statement. This column is applicable only for static statements. The default value of -1 indicates EXPLAIN information that was captured in Version 9 or earlier.

Table 199. DSN_VIEWREF_TABLE description (continued)

Column name	Data type	Description
EXPANSION_REASON	CHAR(2) NOT NULL WITH DEFAULT	This column applies to only statements that reference archive tables or temporal tables. For other statements, this column is blank. Indicates the effect of the CURRENT TEMPORAL BUSINESS_TIME special register, the CURRENT TEMPORAL SYSTEM_TIME special register, and the SYSIBMADM.GET_ARCHIVE built-in global variable. These items are controlled by the BUSTIMESENSITIVE, SYSTIMESENSITIVE, and ARCHIVESENSITIVE bind options. DB2 implicitly adds certain syntax to the query if one of the following conditions are true: • The SYSIBMADM.GET_ARCHIVE global variable is set to Y and the ARCHIVESENSITIVE bind option is set to YES • The CURRENT TEMPORAL BUSINESS_TIME special register is not null and the BUSTIMESENSITIVE bind option is set to YES • The CURRENT TEMPORAL SYSTEM_TIME special register is not null and the SYSTIMESENSITIVE bind option is set to YES This column can have one of the following values: **A** The query contains implicit query transformation as a result of the SYSIBMADM.GET_ARCHIVE built-in global variable. **B** The query contains implicit query transformation as a result of the CURRENT TEMPORAL BUSINESS_TIME special register. **S** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register. **SB** The query contains implicit query transformation as a result of the CURRENT TEMPORAL SYSTEM_TIME special register and the CURRENT TEMPORAL BUSINESS_TIME special register. **blank** The query does not contain implicit query transformation.

PSPI

Tables that are used by accelerators

To interact with accelerator servers, DB2 requires two tables that record characteristics of those accelerator servers: SYSACCELERATORS and SYSACCELERATEDTABLES.

The following table lists the table space and indexes for these two tables, and lists the index fields for each index. The indexes are in ascending order, except where noted.

Table 200. Table spaces and indexes for the tables that are used for accelerators

TABLE SPACE DSNACCEL. ...	TABLE SYSACCEL. ...	INDEX SYSACCEL. ...	INDEX FIELDS
SYSACCEL	SYSACCELERATORS	DSNACC01	ACCELERATORNAME
	SYSACCELERATEDTABLES	DSNACT01	CREATOR.NAME.ACCELERATORNAME

SYSACCEL.SYSACCELERATORS table

The SYSACCEL.SYSACCELERATORS table contains rows that describe the characteristics of each accelerator server.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
ACCELERATORNAME	VARCHAR(128) NOT NULL	A unique name for the accelerator server. This is the name by which the accelerator server is known to the local DB2 accelerated query tables.	G
LOCATION	VARCHAR(128)	Identifies the location name that is associated with the accelerator server.	G

SYSACCEL.SYSACCELERATEDTABLES table

The SYSACCEL.SYSACCELERATEDTABLES table contains rows that describe the characteristics of each table that is marked for acceleration.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
NAME	VARCHAR(128) NOT NULL	The name of the table.	G
CREATOR	VARCHAR(128) NOT NULL	The schema of the table.	G
ACCELERATORNAME	VARCHAR(128) NOT NULL	A unique name for the accelerator server. This is the name by which the accelerator server is known to the local DB2 accelerated query tables.	G
REMOTENAME	VARCHAR(128) NOT NULL	The name of the base alias object.	G
REMOTECREATOR	VARCHAR(128) NOT NULL	The owner of the base alias object.	G
ENABLE	CHAR(1) NOT NULL	Indicates whether the remote table is enabled or disabled for query acceleration: **Y** Enabled **N** Disabled	G
CREATEDBY	VARCHAR(128) NOT NULL	The primary authorization ID of the user who created the table.	G
CREATEDTS	TIMESTAMP NOT NULL WITH DEFAULT	The time when the CREATE statement was executed for the table.	G
ALTEREDTS	TIMESTAMP NOT NULL WITH DEFAULT	The time when the table was last altered.	G
REFRESH_TIME	TIMESTAMP NOT NULL WITH DEFAULT	The timestamp when the data was last refreshed. If the data was not refreshed, this column contains the default timestamp ('0001-01-01.00.00.00.000000').	G
SUPPORTLEVEL	SMALLINT NOT NULL	Internal use only.	I
ARCHIVE	CHAR(1)	The archive status of the table in the accelerator database: **A** The table is archived in the accelerator server that is specified by the ACCELERATORNAME value. The accelerator server contains active and archived data. **C** The table is archived in other accelerator servers. The accelerator server that is specified by the ACCELERATORNAME value contains only active data. **blank** The table is not archived in an accelerator server.	G

Tables that are used for program authorization

For program authorization, a table is provided to record the authorization for a program to execute a plan.

Table spaces and indexes for program authorization

Tables that are used for program authorization are contained in certain table spaces and have indexes.

The following table lists the table space and index for the table that is used for program authorization, and lists the index fields for the index. The index is in ascending order.

Table 201. Table spaces and indexes for the tables that are used for program authorization

TABLE SPACE DSNMDCDB. ...	TABLE SYSIBM. ...	INDEX SYSIBM. ...	INDEX FIELDS
DSNMDCTS	DSNPROGAUTH	DSNPROGAUTH_IDX1	PROGNAME.PLANNAME

SYSIBM.DSNPROGAUTH table

The SYSIBM.DSNPROGAUTH table enables program authorization with or without program data integrity checking.

Column name	Data type	Description	Use
PROGNAME	VARCHAR(24) NOT NULL	Name of the application program that can run the plan.	G
PLANNAME	VARCHAR(24) NOT NULL	Name of the application plan for the application program.	G
PROGMDCVAL	CHAR(16) NOT NULL FOR BIT DATA WITH DEFAULT X'0000000000000000- 00000000000000000'	Reserved.	G
PROGMDCPAD	CHAR(1) NOT NULL WITH DEFAULT	Reserved.	G
CREATOR	VARCHAR(128) NOT NULL WITH DEFAULT CURRENT SQLID	The authorization ID under which the row was inserted or most recently updated.	G
ENABLED	CHAR(1) NOT NULL WITH DEFAULT 'N'	Whether program authorization is enabled: **Y** Program authorization is enabled. **N** Program authorization is disabled.	
CREATETS	TIMESTAMP NOT NULL WITH DEFAULT	The time at which the row was inserted or most recently updated.	G
REMARKS	VARCHAR(762)	A user-specified character string.	G

Using the catalog in database design

Retrieving information from the catalog by using SQL statements, can be helpful in designing your relational database.

GUPI

DB2 SQL Reference lists all the DB2 catalog tables and the information stored in them.

The information in the catalog is vital to normal DB2 operation. You can *retrieve* catalog information, but *changing* it can have serious consequences. Therefore you cannot execute insert or delete operations that affect the catalog, and only a limited number of columns exist that you can update. Exceptions to these restrictions are the SYSIBM.SYSSTRINGS, SYSIBM.SYSCOLDIST, and SYSIBM.SYSCOLDISTSTATS catalog tables, into which you can insert rows and proceed to update and delete rows.

To retrieve information from the catalog, you need at least the SELECT privilege on the appropriate catalog tables.

Note: Some catalog queries can result in long table space scans.

GUPI

Retrieving catalog information about DB2 storage groups

The SYSIBM.SYSSTOGROUP and SYSIBM.SYSVOLUMES tables contain information about DB2 storage groups and the volumes in those storage groups.

Procedure

GUPI

To obtain information about DB2 storage groups and the volumes in those storage groups:

Query the SYSIBM.SYSSTOGROUP and SYSIBM.SYSVOLUMES tables. The following query shows what volumes are in a DB2 storage group, how much space is used, and when that space was last calculated.

```
SELECT SGNAME,VOLID,SPACE,SPCDATE
  FROM SYSIBM.SYSVOLUMES,SYSIBM.SYSSTOGROUP
  WHERE SGNAME=NAME
  ORDER BY SGNAME;
```

GUPI

Related reference:

"SYSIBM.SYSSTOGROUP table" on page 2377

"SYSIBM.SYSVOLUMES table" on page 2441

Retrieving catalog information about a table

The SYSIBM.SYSTABLES table contains information about every table, view, and alias in your DB2 system.

About this task

GUPI The SYSIBM.SYSTABLES table contains a row for every table, view, and alias in your DB2 system. Each row tells you whether the object is a table, a view, or an alias, its name, who created it, what database it belongs to, what table space it belongs to, and other information. The SYSTABLES table also has a REMARKS column in which you can store your own information about the table in question.

Procedure

To retrieve catalog information about a table:

Query the SYSIBM.SYSTABLES table. The following example query displays all the information for the project activity sample table:

```
SELECT *
  FROM SYSIBM.SYSTABLES
  WHERE NAME = 'PROJACT'
  AND CREATOR = 'DSN8B10';
```

GUPI

Related concepts:

"Adding and retrieving comments" on page 2606

Related reference:

"SYSIBM.SYSTABLES table" on page 2396

Retrieving catalog information about partition order

The LOGICAL_PART column in the SYSIBM.SYSTABLEPART table contains information for key order or logical partition order.

Procedure

GUPI

To retrieve catalog information about partition order:

Query the SYSIBM.SYSTABLEPART table. The following statement displays information on partition order in ascending limit value order:

```
SELECT LIMITKEY, PARTITION
  FROM SYSIBM.SYSTABLEPART
  ORDER BY LOGICAL_PART;
```

GUPI

Related reference:

"SYSIBM.SYSTABLEPART table" on page 2387

Retrieving catalog information about aliases

Query SYSIBM.SYSTABLES to obtain information about aliases.

About this task

GUPI

You can use the SYSIBM.SYSTABLES table to find information about aliases by referencing the following three columns:

- LOCATION contains your subsystem's location name for the remote system, if the object on which the alias is defined resides at a remote subsystem.
- TBCREATOR contains the schema table or view.
- TBNAME contains the name of the table or the view.

You can also find information about aliases by using the following user-defined functions:

- TABLE_NAME returns the name of a table, view, or undefined object found after resolving aliases for a user-specified object.
- TABLE_SCHEMA returns the schema name of a table, view, or undefined object found after resolving aliases for a user-specified object.
- TABLE_LOCATION returns the location name of a table, view, or undefined object found after resolving aliases for a user-specified object.

The NAME and CREATOR columns of the SYSTABLES table contain the name and schema of the alias, and three other columns contain the following information for aliases:

- TYPE is A.

- DBNAME is DSNDB06.
- TSNAME is SYSTSTAB.

If similar tables at different locations have names with the same second and third parts, you can retrieve the aliases for them with a query like this one:

```
SELECT LOCATION, CREATOR, NAME
  FROM SYSIBM.SYSTABLES
    WHERE TBCREATOR='DSN8B10' AND TBNAME='EMP'
      AND TYPE='A';
```

GUPI

Related reference:

"SYSIBM.SYSTABLES table" on page 2396

"TABLE_NAME" on page 2631

"TABLE_SCHEMA" on page 2633

"TABLE_LOCATION" on page 2629

Retrieving catalog information about columns

The SYSIBM.SYSCOLUMNS table has one row for each column of every table and view.

Procedure

GUPI

To obtain information about the columns of a table or view:

Query the SYSIBM.SYSCOLUMNS table.
The following statement retrieves information about columns in the sample department table:

```
SELECT NAME, TBNAME, COLTYPE, LENGTH, NULLS, DEFAULT
  FROM SYSIBM.SYSCOLUMNS
  WHERE TBNAME='DEPT'
  AND TBCREATOR = 'DSN8B10';
```

The result is shown below; for each column, the following information about each column is given:

- The column name
- The name of the table that contains it
- Its data type
- Its length attribute. For LOB columns, the LENGTH column shows the length of the pointer to the LOB.
- Whether it allows nulls
- Whether it allows default values

```
NAME      TBNAME  COLTYPE         LENGTH  NULLS  DEFAULT
DEPTNO    DEPT    CHAR               3    N       N
DEPTNAME  DEPT    VARCHAR           36    N       N
MGRNO     DEPT    CHAR               6    Y       N
ADMRDEPT  DEPT    CHAR               3    N       N
```

GUPI

Related tasks:

"Retrieving catalog information about LOBs" on page 2604

Related reference:

"SYSIBM.SYSCOLUMNS table" on page 2155

Retrieving catalog information about indexes

The SYSIBM.SYSINDEXES table contains a row for every index, including indexes on catalog tables.

Procedure

GUPI

To obtain information about indexes:

Query the SYSIBM.SYSINDEXES table. For example, to retrieve a row about an index named XEMPL2:

```
SELECT *
  FROM SYSIBM.SYSINDEXES
  WHERE NAME = 'XEMPL2'
  AND CREATOR = 'DSN8B10';
```

A table can have more than one index. To display information about all the indexes of a table:

```
SELECT *
  FROM SYSIBM.SYSINDEXES
  WHERE TBNAME = 'EMP'
  AND TBCREATOR = 'DSN8B10';
```

GUPI

Related reference:

"SYSIBM.SYSINDEXES table" on page 2211

Retrieving catalog information about views

For every view you create, DB2 stores descriptive information in several catalog tables. Query these catalog tables to obtain information about views in your database.

About this task

GUPI

The following actions occur in the catalog after the execution of CREATE VIEW:

- A row is inserted into the SYSIBM.SYSTABLES table.
- A row is inserted into the SYSIBM.SYSTABAUTH table to record the owner's privileges on the view.
- For each column of the view, a row is inserted into the SYSIBM.SYSCOLUMNS table.
- One or more rows are inserted into the SYSIBM.SYSVIEWS table to record the text of the CREATE VIEW statement.
- For each table or view on which the view is dependent, a row is inserted into the SYSIBM.SYSVIEWDEP table to record the dependency.

GUPI

Procedure

To obtain information about views:

Query one or more catalog tables.

Related reference:

"CREATE VIEW" on page 1527
"SYSIBM.SYSTABLES table" on page 2396
"SYSIBM.SYSTABAUTH table" on page 2383
"SYSIBM.SYSCOLUMNS table" on page 2155
"SYSIBM.SYSVIEWS table" on page 2437
"SYSIBM.SYSVIEWDEP table" on page 2436

Retrieving catalog information about authorizations

The SYSIBM.SYSTABAUTH table contains information about who can access your data.

Procedure

To obtain information about who can access your data:

GUPI Query the SYSIBM.SYSTABAUTH table. The following query retrieves the names of all users who have been granted access to the DSN8B10.DEPT table.

```
SELECT GRANTEE
  FROM SYSIBM.SYSTABAUTH
  WHERE TTNAME = 'DEPT'
    AND GRANTEETYPE <> 'P'
    AND TCREATOR = 'DSN8B10';
```

GRANTEE is the name of the column that contains authorization IDs for users of tables. The TTNAME and TCREATOR columns specify the DSN8B10.DEPT table. The clause GRANTEETYPE <> 'P' ensures that you retrieve the names only of users (not application plans or packages) that have authority to access the table.

GUPI

Related reference:

"SYSIBM.SYSTABAUTH table" on page 2383

Retrieving catalog information about primary keys

The SYSIBM.SYSCOLUMNS table identifies columns of a primary key in column KEYSEQ; a nonzero value indicates the place of a column in the primary key.

Procedure

GUPI

To obtain catalog information about primary keys:

Query the SYSIBM.SYSCOLUMNS table. To retrieve the creator, database, and names of the columns in the primary key of the sample project activity table using SQL statements, execute:

```
SELECT TBCREATOR, TBNAME, NAME, KEYSEQ
  FROM SYSIBM.SYSCOLUMNS
  WHERE TBCREATOR = 'DSN8B10'
  AND TBNAME = 'PROJACT'
  AND KEYSEQ > 0
    ORDER BY KEYSEQ;
```

The SYSIBM.SYSINDEXES table identifies the primary index of a table by the value P in column UNIQUERULE. To find the name, creator, database, and index space of the primary index on the project activity table, execute:

```
SELECT TBCREATOR, TBNAME, NAME, CREATOR, DBNAME, INDEXSPACE
  FROM SYSIBM.SYSINDEXES
  WHERE TBCREATOR = 'DSN8B10'
  AND TBNAME = 'PROJACT'
  AND UNIQUERULE = 'P';
```

GUPI

Related reference:

"SYSIBM.SYSCOLUMNS table" on page 2155

"SYSIBM.SYSINDEXES table" on page 2211

Retrieving catalog information about foreign keys

The SYSIBM.SYSRELS and SYSIBM.SYSFOREIGNKEYS tables contain information about referential constraints and the columns of the foreign key that defines the constraint.

About this task

GUPI

The SYSIBM.SYSRELS table contains information about referential constraints, and each constraint is uniquely identified by the schema and name of the dependent table and the constraint name (RELNAME). The SYSIBM.SYSFOREIGNKEYS table contains information about the columns of the foreign key that defines the constraint.

Procedure

To obtain information about referential constraints and the columns of the foreign key that defines the constraint:

Query the SYSIBM.SYSRELS table or the SYSIBM.SYSFOREIGNKEYS table. To retrieve the constraint name, column names, and parent table names for every relationship in which the project table is a dependent, execute:

```
SELECT A.CREATOR, A.TBNAME, A.RELNAME, B.COLNAME, B.COLSEQ,
       A.REFTBCREATOR, A.REFTBNAME
  FROM SYSIBM.SYSRELS A, SYSIBM.SYSFOREIGNKEYS B
  WHERE A.CREATOR = 'DSN8B10'
  AND B.CREATOR = 'DSN8B10'
  AND A.TBNAME = 'PROJ'
  AND B.TBNAME = 'PROJ'
  AND A.RELNAME = B.RELNAME
    ORDER BY A.RELNAME, B.COLSEQ;
```

To find information about the foreign keys of tables to which the project table is a parent:

```
SELECT A.RELNAME, A.CREATOR, A.TBNAME, B.COLNAME, B.COLNO
  FROM SYSIBM.SYSRELS A, SYSIBM.SYSFOREIGNKEYS B
  WHERE A.REFTBCREATOR = 'DSN8B10'
  AND A.REFTBNAME = 'PROJ'
  AND A.RELNAME = B.RELNAME
    ORDER BY A.RELNAME, B.COLNO;
```

GUPI

Related reference:

"SYSIBM.SYSRELS table" on page 2339

"SYSIBM.SYSFOREIGNKEYS table" on page 2209

Retrieving catalog information about check pending

The SYSIBM.SYSTABLESPACE table contains information about table spaces that are in check-pending status.

About this task

GUPI

The SYSIBM.SYSTABLESPACE table indicates that a table space is in check-pending status by a value in column STATUS: P if the entire table space has that status, S if the status has a scope of less than the entire space.

Procedure

To obtain information about table spaces that are in check-pending status:

Query the SYSIBM.SYSTABLESPACE table. To list all table spaces whose use is restricted for *any* reason, issue this command:

```
-DISPLAY DATABASE (*) SPACENAM(*) RESTRICT
```

To retrieve the names of table spaces in check-pending status only, with the names of the tables they contain, execute:

```
SELECT A.DBNAME, A.NAME, B.CREATOR, B.NAME
  FROM SYSIBM.SYSTABLESPACE A, SYSIBM.SYSTABLES B
  WHERE A.DBNAME = B.DBNAME
  AND A.NAME = B.TSNAME
  AND (A.STATUS = 'P' OR A.STATUS = 'S')
    ORDER BY 1, 2, 3, 4;
```

GUPI

Related reference:

"SYSIBM.SYSTABLESPACE table" on page 2404

Retrieving catalog information about check constraints

The SYSIBM.SYSCHECKS and SYSIBM.SYSCHECKDEP tables contain information about check constraints.

About this task

GUPI

Information about check constraints is stored in the DB2 catalog in:

- SYSIBM.SYSCHECKS, which contains one row for each check constraint defined on a table
- SYSIBM.SYSCHECKDEP, which contains one row for each reference to a column in a check constraint

Procedure

To retrieve catalog information about check constraints:

Query the SYSIBM.SYSCHECKS and SYSIBM.SYSCHECKDEP tables. The following query shows all check constraints on all tables named SIMPDEPT and SIMPEMPL in order by column name within table schema. It shows the name, authorization ID of the creator, and text for each constraint. A constraint that uses more than one column name appears more than once in the result.

```
CREATE TABLE SIMPDEPT
  (DEPTNO   CHAR(3) NOT NULL,
   DEPTNAME VARCHAR(12) CONSTRAINT CC1 CHECK (DEPTNAME IS NOT NULL),
   MGRNO    CHAR(6),
   MGRNAME  CHAR(6));
```

```
SELECT A.TBOWNER, A.TBNAME, B.COLNAME,
A.CHECKNAME, A.CREATOR, A.CHECKCONDITION
FROM SYSIBM.SYSCHECKS A, SYSIBM.SYSCHECKDEP B
WHERE A.TBOWNER = B.TBOWNER
  AND A.TBNAME = B.TBNAME
  AND B.TBNAME = 'SIMPDEPT'
  AND A.CHECKNAME = B.CHECKNAME
  ORDER BY TBOWNER, TBNAME, COLNAME;
```

GUPI

Related reference:

"SYSIBM.SYSCHECKS table" on page 2143

"SYSIBM.SYSCHECKDEP table" on page 2142

Retrieving catalog information about LOBs

The SYSIBM.SYSAUXRELS table contains information about the relationship between a base table and an auxiliary table.

Procedure

GUPI

To retrieve catalog information about LOBs:

Query the SYSIBM.SYSAUXRELS table. For example, this query returns information about the name of the LOB columns for the employee table and its associated auxiliary table schema and name:

```
SELECT COLNAME, PARTITION, AUXTBOWNER, AUXTBNAME
   FROM SYSIBM.SYSAUXRELS
   WHERE TBNAME = 'EMP' AND TBOWNER = 'DSN8B10';
```

Information about the length of a LOB is in the LENGTH2 column of the SYSCOLUMNS table. You can query information about the length of the column as it is returned to an application with the following query:

```
SELECT NAME, TBNAME, COLTYPE, LENGTH2, NULLS, DEFAULT
  FROM SYSIBM.SYSCOLUMNS
  WHERE TBNAME='DEPT'
  AND TBCREATOR = 'DSN8B10';
```

GUPI

Related reference:

"SYSIBM.SYSAUXRELS table" on page 2141

"SYSIBM.SYSCOLUMNS table" on page 2155

Retrieving catalog information about user-defined functions and stored procedures

The SYSIBM.SYSROUTINES table contains information about routines.

Procedure

To retrieve information about user-defined functions and stored procedures:

GUPI Query the SYSIBM.SYSROUTINES table to obtain information about user-defined functions and stored procedures. You can use this example to find packages with stored procedures that were created prior to Version 6 and then migrated to the SYSIBM.SYSROUTINES table:

```
SELECT SCHEMA, NAME FROM SYSIBM.SYSROUTINES
   WHERE ROUTINETYPE = 'P';
```

You can use this query to retrieve information about user-defined functions:

```
SELECT SCHEME, NAME, FUNCTION_TYPE, PARM_COUNT FROM SYSIBM.SYSROUTINES
   WHERE ROUTINETYPE='F';
```

GUPI

Related tasks:

Preparing a client program that calls a remote stored procedure (DB2 Application programming and SQL)

Related reference:

"SYSIBM.SYSROUTINES table" on page 2346

Retrieving catalog information about triggers

The SYSIBM.SYSTRIGGERS table contains information about triggers.

Procedure

To retrieve catalog information about triggers:

GUPI Query the SYSIBM.SYSTRIGGERS table to obtain information about the triggers defined in your databases. You can issue this query to find all the triggers defined on a particular table, their characteristics, and to determine the order they are activated in:

```
SELECT DISTINCT SCHEMA, NAME, TRIGTIME, TRIGEVENT, GRANULARITY, CREADEDTS
   FROM SYSIBM.SYSTRIGGERS
   WHERE TBNAME = 'EMP' AND TBOWNER = 'DSN8B10';
```

Issue this query to retrieve the text of a particular trigger:

```
SELECT STATEMENT, CREATEDTS
   FROM SYSIBM.SYSTRIGGERS
   WHERE SCHEMA = schema_name
      AND NAME = trigger_name
   ORDER BY CREATEDTS;
```

Issue this query to determine triggers that must be rebound because they are invalidated after objects are dropped or altered:

```
SELECT COLLID, NAME
   FROM SYSIBM.SYSPACKAGE
   WHERE TYPE = 'T'
      AND (VALID = 'N' OR OPERATIVE = 'N');
```

GUPI

Related reference:

"SYSIBM.SYSTRIGGERS table" on page 2422

Retrieving catalog information about sequences

The SYSIBM.SYSSEQUENCES and SYSIBM.SYSSEQUENCEAUTH tables contain information about sequences.

Procedure

GUPI

To obtain information about sequences:

Query the SYSIBM.SYSSEQUENCES or SYSIBM.SYSSEQUENCEAUTH table. To retrieve the attributes of a sequence, issue this query:

```
SELECT *
  FROM SYSIBM.SYSSEQUENCES
  WHERE NAME = 'MYSEQ' AND SCHEMA = 'USER1B';
```

Issue this query to determine the privileges that user USER1B has on sequences:

```
SELECT GRANTOR, NAME, DATEGRANTED, ALTERAUTH, USEAUTH
  FROM SYSIBM.SEQUENCEAUTH
  WHERE GRANTEE = 'USER1B';
```

GUPI

Related reference:

"SYSIBM.SYSSEQUENCES table" on page 2366

"SYSIBM.SYSSEQUENCEAUTH table" on page 2364

Adding and retrieving comments

After you create an object, you can provide explanatory information about it for future reference. For example, you can provide information about the purpose of the object, who uses it, and anything unusual about it.

GUPI

You can create comments about tables, views, indexes, aliases, packages, plans, distinct types, triggers, stored procedures, and user-defined functions. You can store a comment about the table or the view as a whole, and you can also include a comment for *each column*. A comment must not exceed 762 bytes.

A comment is especially useful if your names do not clearly indicate the contents of columns or tables. In that case, use a comment to describe the specific contents of the column or table.

Below are two examples of COMMENT:

```
COMMENT ON TABLE DSN8B10.EMP IS
  'Employee table. Each row in this table represents one
   employee of the company.';
COMMENT ON COLUMN DSN8B10.PROJ.PRSTDATE IS
  'Estimated project start date. The format is DATE.';
```

After you execute a COMMENT statement, your comments are stored in the REMARKS column of SYSIBM.SYSTABLES or SYSIBM.SYSCOLUMNS. (Any comment that is already present in the row is replaced by the new one.) The next two examples retrieve the comments that are added by the previous COMMENT statements.

```
SELECT REMARKS
  FROM SYSIBM.SYSTABLES
  WHERE NAME = 'EMP'
  AND CREATOR = 'DSN8B10';
SELECT REMARKS
  FROM SYSIBM.SYSCOLUMNS
  WHERE NAME = 'PRSTDATE' AND TBNAME = 'PROJ'
  AND TBCREATOR = 'DSN8B10';
```

GUPI

Verifying the accuracy of the database definition

You can use the catalog to verify the accuracy of your database definition.

Procedure

GUPI To verify that you have created the objects in your database and check that no errors are in your CREATE statements:

Query the catalog tables to verify that your tables are in the correct table space, your table spaces are in the correct storage group, and so on. GUPI

Related reference:

"DB2 catalog tables" on page 2102

Sample user-defined functions

Some sample user-defined functions are provided with DB2. You can use the functions in your applications just as you would use other user-defined functions, or as examples to help you define your own user-defined functions..

- To use these functions in your applications: Use the functions only if installation job DSNTEJ2U, which prepares the functions for use, has been run. Because the external programs that implement the logic of the sample functions are written

in C and C++, the installation job requires that your site has IBM C/C++ for OS/390®. For information on installation job DSNTEJ2U, see *DB2 Installation Guide*.

- If you want to use these functions as examples to help you define and implement your own user-defined functions: Data set *prefix*.SDSNSAMP contains the code for the sample functions.

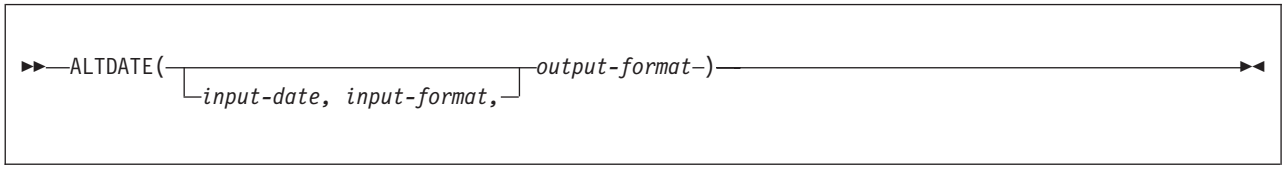
The following table lists the sample user-defined functions. The detailed descriptions of the functions include their external program names and specific names. The functions are in schema DSN8. The functions are defined to treat character or graphic string parameters, both input and output, as EBCDIC-encoded data.

Table 202. DB2 sample user-defined functions

Function Name	Description
ALTDATE	Returns the current date or a user-specified date in a user-specified format
ALTTIME	Returns the current time or a user-specified time in a user-specified format
CURRENCY	Returns a floating-point number as a currency value
DAYNAME	Returns the name of the day of the week on which a date in ISO format falls
MONTHNAME	Returns the name of the month in which a date in ISO format falls
TABLE_LOCATION	Returns the location name of a table or view after resolving any aliases
TABLE_NAME	Returns the unqualified name of a table or view after resolving any aliases
TABLE_SCHEMA	Returns the schema name of a table or view after resolving any aliases
WEATHER	Shows how to use a user-defined table function to make non-relational data available for SQL manipulation

ALTDATE

The ALTDATE function returns the current date in the specified format or converts a user-specified date from one format to another.



The schema is DSN8.

The ALTDATE function returns the current date in one of the following formats or converts a user-specified date from one format to another:

```
D MONTH YY    D MONTH YYYY    DD MONTH YY    DD MONTH YYYY
D.M.YY        D.M.YYYY        DD.MM.YY       DD.MM.YYYY
D-M-YY        D-M-YYYY        DD-MM-YY       DD-MM-YYYY
D/M/YY        D/M/YYYY        DD/MM/YY       DD/MM/YYYY
M/D/YY        M/D/YYYY        MM/DD/YY       MM/DD/YYYY
YY/M/D        YYYY/M/D        YY/MM/DD       YYYY/MM/DD
YY.M.D        YYYY.M.D        YY.MM.DD       YYYY.MM.DD
              YYYY-M-D                       YYYY-MM-DD
              YYYY-D-XX                      YYYY-DD-XX
              YYYY-XX-D                      YYYY-XX-DD
where:
D:     Suppress leading zero if the day is less than 10
DD:    Retain leading zero if the day is less than 10
M:     Suppress leading zero if the month is less than 10
MM:    Retain leading zero if the month is less than 10
MONTH: Use English-language name of month
XX:    Use a capital Roman numeral for month
YY:    Use a year format without century
YYYY:  Use a year format with century
```

The ALTDATE function demonstrates how you can create an overloaded function—a function name for which there are multiple function instances. Each instance supports a different parameter list enabling you to group related but distinct functions in a single user-defined function. The ALTDATE function has two forms.

Form 1: ALTDATE(*output-format***)**

This form of the function converts the current date into the specified format.

output-format

A character string that matches one of the 34 date formats that are shown above. The character string must have a data type of VARCHAR and an actual length that is not greater than 13 bytes.

The result of the function is VARCHAR(17).

Form 2: ALTDATE(*input-date, input-format, output-format***)**

This form of the function converts a date (*input-date*) in one user-specified format (*input-format*) into another format (*output-format*).

input-date

The argument must be a date or a character string representation of a date

in the format specified by *input-format*. The character string must have a data type of VARCHAR and an actual length that is not greater than 17 bytes.

input-format
A character string that matches one of the 34 date formats that are shown above. The character string must have a data type of VARCHAR and an actual length that is not greater than 13 bytes.

output-format
A character string that matches one of the 34 date formats that are shown above. The character string must have a data type of VARCHAR and an actual length that is not greater than 13 bytes.

The result of the function is VARCHAR(17).

The following table shows the external and specific names for the two forms of the function, which are based on the input to the function.

Table 203. External program and specific names for ALTDATE

Conversion type	Input arguments	External name	Specific name
Current date	*output-format* (VARCHAR)	DSN8DUAD	DSN8.DSN8DUADV
User-specified date	*input-date* (VARCHAR) *input-format* (VARCHAR) *output-format* (VARCHAR)	DSN8DUCD	DSN8.DSN8DUCDVVV
	input-date (DATE) *input-format* (VARCHAR) *output-format* (VARCHAR)	DSN8DUCD	DSN8.DSN8DUCDDVV

Example 1: Convert the current date into format 'DD MONTH YY', a format that will include any leading zero for the month, the name of the month in English, and the year without the two digits for the century.

```
VALUES DSN8.ALTDATE( 'DD MONTH YY' );
```

Example 2: Convert the current date into format 'D.M.YYYY', a format that will suppress any leading zero for the day or month and include the year with the century.

```
VALUES DSN8.ALTDATE( 'D.M.YYYY' );
```

Example 3: Convert the current date into format 'YYYY-XX-DD', a format that will include the century, the month of the year as a roman numeral, and the day of the month with any leading zero.

```
VALUES DSN8.ALTDATE( 'YYYY-XX-DD' );
```

Example 4: Convert a date in the format of 'DD MONTH YYYY' to a date in the format of 'YYYY/MM/DD'.

```
VALUES DSN8.ALTDATE( '11 November 1918',
                     'DD MONTH YYYY',
                     'YYYY/MM/DD' );
```

The result of the above example is '1918/11/18'.

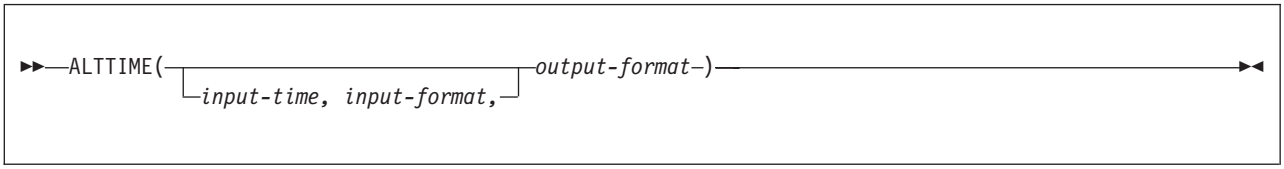
Example 5: Convert the date that employee 000130 was hired, a date in ISO format, into the format of 'D.M.YY'.

```
SELECT  FIRSTNME || ' '
     ||  LASTNAME || ' was hired on '
     ||  DSN8.ALTDATE( HIREDATE,
                       'YYYY-MM-DD',
                       'D.M.YY' )
  FROM  EMP
 WHERE  EMPNO  = '000130';
```

Assuming that the HIREDATE is '1971-07-28', the above example returns: 'DELORES QUINTANA was hired on 28.7.71'.

ALTTIME

The ALTTIME function returns the current time in the specified format or converts a user-specified time from one format to another.



The schema is DSN8.

The ALTTIME function returns the current time in one of the following formats or converts a user-specified time from one of the formats to another:

```
H:MM AM/PM           HH:MM AM/PM
HH:MM:SS AM/PM       HH:MM:SS
H.MM                 HH.MM
H.MM.SS              HH.MM.SS
where:
H:      Suppress leading zero if the hour is less than 10
HH:     Retain leading zero if the hour is less than 10
M:      Suppress leading zero if the minute is less than 10
MM:     Retain leading zero if the minute is less than 10
AM/PM:  Return time in 12-hour clock format, else 24-hour
```

The ALTTIME function demonstrates how you can create an overloaded function—a function name for which there are multiple function instances. Each instance supports a different parameter list enabling you to group related but distinct functions in a single user-defined function. The ALTTIME function has two forms.

Form 1: ALTTIME(*output-format***)**
This form of the function converts the current time into the specified format.

output-format
A character string that matches one of the 8 time formats that are shown above. The character string must have a data type of VARCHAR and an actual length that is not greater than 14 bytes.

The result of the function is VARCHAR(11).

Form 2: ALTTIME(*input-time, input-format, output-format***)**
This form of the function converts a time (*input-date*) in one user-specified format (*input-format*) into another format (*output-format*).

input-time
The argument must be a time or a character string representation of a time in the format specified by *input-format*. A character string argument must have a data type of VARCHAR and an actual length that is not greater than 11 bytes.

input-format
A character string that matches one of the 8 time formats that are shown above. The character string must have a data type of VARCHAR and an actual length that is not greater than 14 bytes.

output-format
A character string that matches one of the 8 time formats that are shown

above. The character string must have a data type of VARCHAR and an actual length that is not greater than 14 bytes.

The result of the function is VARCHAR(11).

The following table shows the external program and specific names for the two forms of the function, which are based on the input to the function.

Table 204. External and specific names for ALTTIME

Conversion type	Input arguments	External name	Specific name
Current time	*output-format* (VARCHAR)	DSN8DUAT	DSN8.DSN8DUATV
User-specified time	*input-time* (VARCHAR) *input-format* (VARCHAR) *output-format* (VARCHAR)	DSN8DUCT	DSN8.DSN8DUCTVVV
	input-time (TIME) *input-format* (VARCHAR) *output-format* (VARCHAR)	DSN8DUCT	DSN8.DSN8DUCTTVV

Example 1: Convert the current time into a 12-hour clock format without seconds, 'H.MM AM/PM'.

```
VALUES DSN8.ALTTIME( 'H:MM AM/PM' );
```

Example 2: Convert the current time into a 24-hour clock format without seconds, 'HH.MM'.

```
VALUES DSN8.ALTTIME( 'HH.MM' );
```

Example 3: Convert the current time into a 24-hour clock format with seconds, 'HH.MM.SS'.

```
VALUES DSN8.ALTTIME( 'HH.MM.SS' );
```

Example 4: Convert '00:00:00', a time in 24-hour clock format with seconds, to a time in 12-hour clock format without seconds.

```
VALUES DSN8.ALTTIME( '00:00:00','HH:MM:SS','HH:MM AM/PM' );
```

The function returns '12:00 AM'.

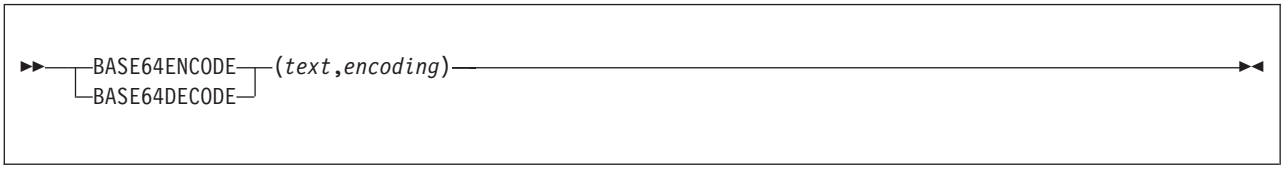
Example 5: Convert '00:00:00', a time in 24-hour clock format with seconds, to a time in 12-hour clock format without seconds and without any leading zero on the hour.

```
VALUES DSN8.ALTTIME( '06.42.37','HH.MM.SS','H:MM AM/PM' );
```

The function returns '6:42 AM'.

BASE64ENCODE and BASE64DECODE

The BASE64ENCODE and BASE64DECODE helper REST functions complete Base64 encoding or decoding of the provided text or data.



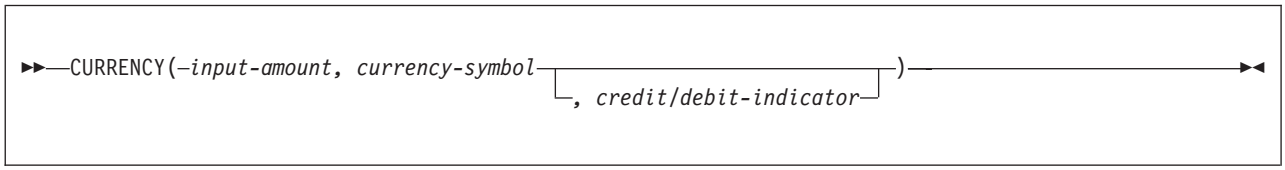
The schema is DB2XML.

text
Specifies the text to encode or decode. For BASE64ENCODE, this argument is provided as a VARCHAR(2732) value and the function returns a Base64-encoded string. For BASE64DECODE, this argument is provided as a Base64-encoded VARCHAR(4096) value and the function returns the data as binary.

encoding
Specifies the character set that is to be used. It can be set to NULL where UTF-8 is used as the default.

CURRENCY

The CURRENCY function returns a value that is formatted as an amount with a user-specified currency symbol and, if specified, one of three symbols that indicate debit or credit.



The schema is DSN8.

input-amount
An expression that specifies the value to be formatted. The expression must be a floating-point value.

currency-symbol
A character string that specifies the currency symbol. The string must have a data type of VARCHAR and an actual length that is not greater than 2 bytes.

credit/debit-indicator
A character string that specifies the symbol that is included with the result to indicate whether the value is negative or positive. The string must have a data type of VARCHAR and an actual length that is not greater than 5 bytes. If *credit/debit-indicator* is not specified or is the value null, the result is formatted without an indicator symbol. You can specify the following symbols:

CR/DB
Bank style. Negative input values are appended with 'DB'; positive input values are appended with 'CR'.

+/- *Arithmetic style.* Negative input values are prefixed with a minus sign '-'; positive values are formatted without symbols.

(/) *Accounting style.* Negative input values are enclosed in parentheses '( )'; positive values are formatted without symbols.

The result of the function is VARCHAR(19).

The CURRENCY function uses the C language functions strfmon to facilitate formatting of money amounts and setlocale to initialize strfmon for local conventions. If setlocale fails, the CURRENCY function returns an error.

The following table shows the external program and specific names for CURRENCY. The specific names differ depending on the input to the function.

Table 205. External program and specific names for CURRENCY

Input arguments	External name	Specific name
input-amount *currency-symbol*	DSN8DUCY	DSN8.DSN8DUCYFV
input-amount *currency-symbol* *credit/debit-indicator*	DSN8DUCY	DSN8.DSN8DUCYFVV

Example 1: Express® '-1234.56' as an amount in US dollars, using the bank style debit/credit indicator to indicate whether the value is negative or positive.

```
VALUES DSN8.CURRENCY( -1234.56,'$','CR/DB' );
```

The result of the function is '$1,234.56 DB'.

Example 2: Express '-1234.56' as an amount in Deutsche marks, using the accounting style debit/credit indicator to indicate whether the value is negative or positive.

```
VALUES DSN8.CURRENCY( -1234.56,'DM','(/)' );
```

The result of the function is '(DM 1,234.56)'.

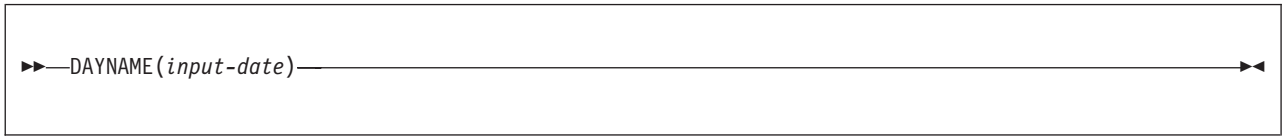
Example 3: Express '-1234.56' as an amount in Canadian dollars, using the accounting style debit/credit indicator to indicate whether the value is negative or positive.

```
VALUES DSN8.CURRENCY( -1234.56,'CD','+/-' );
```

The result of the function is '-CD 1,234.56'.

DAYNAME

The DAYNAME function returns the name of the weekday on which a given date falls. The name is returned in English.



The schema is DSN8.

input-date
A valid date or valid character string representation of a date. A character string representation The string must have a data type of VARCHAR and an actual length that is not greater than 10 bytes. The date must be in ISO format.

The result of the function is VARCHAR(9).

The DAYNAME function uses the IBM C++ classIDate.

The following table shows the external and specific names for DAYNAME. The specific names differ depending on the data type of the input argument.

Table 206. External and specific names for DAYNAME

Input arguments	External name	Specific name
input-date (VARCHAR)	DSN8EUDN	DSN8.DSN8EUDNV
input-date (DATE)	DSN8EUDN	DSN8.DSN8EUDND

Example 1: For the current date, find the day of the week.

```
VALUES DSN8.DAYNAME( CURRENT DATE );
```

Example 2: Find the day of the week on which leap year falls in the year 2008.

```
VALUES DSN8.DAYNAME( '2008-02-29' );
```

The result of the function is 'Friday'.

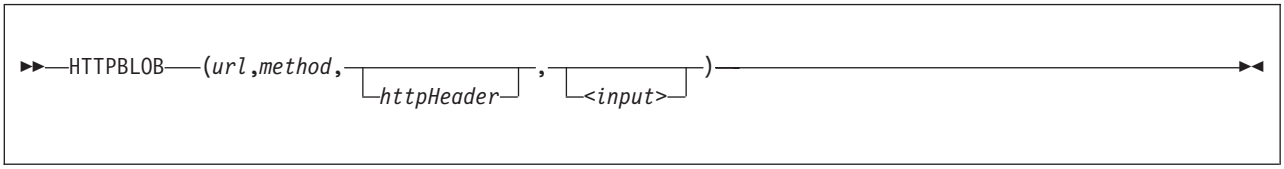
Example 3: Find the day of the week on which Delores Quintana, employee number 000130, was hired.

```
SELECT  FIRSTNME || ' '
    ||  LASTNAME || ' was hired on '
    ||  DSN8.DAYNAME( HIREDATE ) || ', '
    ||  CHAR( HIREDATE )
  FROM  EMP
 WHERE  EMPNO  = '000130';
```

The result of the function is 'DELORES QUINTANA was hired on Wednesday, 1971-07-28'.

HTTPBLOB

The HTTPBLOB REST function completes an HTTP request with the specified HTTP verb. Response messages from the server are returned as BLOB data.



The schema is DB2XML.

url
Specifies the URL at which to complete the request. This argument is defined as a VARCHAR(2048) value.

method
Specifies the HTTP verb to use. Valid values are GET, POST, PUT, and DELETE.

httpHeader
Specifies an optional header XML document. This argument is defined as a CLOB(10K) value.

The XML header document can provide additional HTTP header values in the following format:

```
<httpHeader headerAttribute="headerAtributeValue">
        <header name="name" value="value" />
</httpHeader>
```

headerAttribute
Specify any of the following optional attributes:

connectionTimeout
Specifies an integer value for the connection timeout threshold in milliseconds.

readTimeout
Specifies an integer value for the read timeout threshold in milliseconds.

followRedirects
Specifies whether redirects should be followed. This is a boolean value.

useCaches
Specifies whether caches should be used. This is a boolean value.

headerAttributeValue
Specifies a value for the *headerAttribute*. Separate *headerAttribute* and *headerAttributeValue* combinations with single spaces.

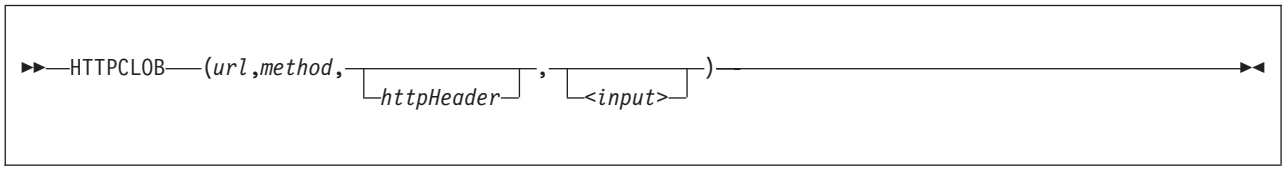
name
The header name.

value
The header value.

<*input***>**
Specifies the data to update at the specified URL. This argument is defined as BLOB(5M).

HTTPCLOB

The HTTPCLOB REST function completes an HTTP request with the specified HTTP verb. Response messages from the server are returned as CLOB data. The character set is converted into the database code page if necessary.



The schema is DB2XML.

url
Specifies the URL at which to complete the request. This argument is defined as a VARCHAR(2048) value.

method
Specifies the HTTP verb to use. Valid values are GET, POST, PUT, and DELETE.

httpHeader
Specifies an optional header XML document. This argument is defined as a CLOB(10K) value.

The XML header document can provide additional HTTP header values in the following format:

```
<httpHeader headerAttribute="headerAtributeValue">
        <header name="name" value="value" />
</httpHeader>
```

headerAttribute
Specify any of the following optional attributes:

connectionTimeout
Specifies an integer value for the connection timeout threshold in milliseconds.

readTimeout
Specifies an integer value for the read timeout threshold in milliseconds.

followRedirects
Specifies whether redirects should be followed. This is a boolean value.

useCaches
Specifies whether caches should be used. This is a boolean value.

headerAttributeValue
Specifies a value for the *headerAttribute*. Separate *headerAttribute* and *headerAttributeValue* combinations with single spaces.

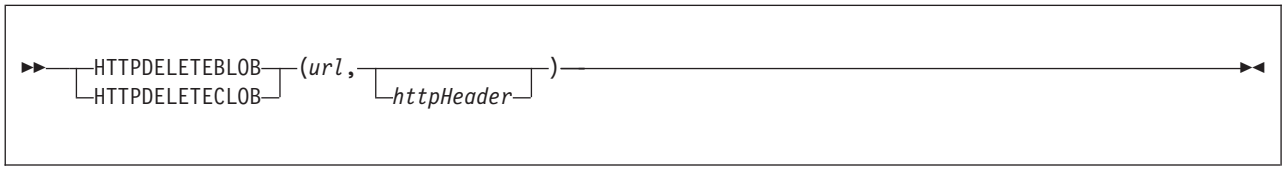
name
The header name.

value
The header value.

| *<input>*
| Specifies the data to update at the specified URL. This argument is defined as
| CLOB(5M).

HTTPDELETEBLOB and HTTPDELETECLOB

The HTTPDELETEBLOB and HTTPDELETECLOB REST functions delete a binary or text-based resource from the specified URL through an HTTP DELETE request. HTTPDELETEBLOB returns messages as BLOB data. HTTPDELETECLOB returns messages as CLOB data. The character set is converted into the database code page if necessary.



The schema is DB2XML.

url
Specifies the URL of the resource being accessed. This parameter is defined as a VARCHAR(2048) value.

httpHeader
Specifies an optional header XML document. This argument is defined as a CLOB(10K) value.

The XML header document can provide additional HTTP header values in the following format:

```
<httpHeader headerAttribute="headerAtributeValue">
        <header name="name" value="value" />
</httpHeader>
```

headerAttribute
Specify any of the following optional attributes:

connectionTimeout
Specifies an integer value for the connection timeout threshold in milliseconds.

readTimeout
Specifies an integer value for the read timeout threshold in milliseconds.

followRedirects
Specifies whether redirects should be followed. This is a boolean value.

useCaches
Specifies whether caches should be used. This is a boolean value.

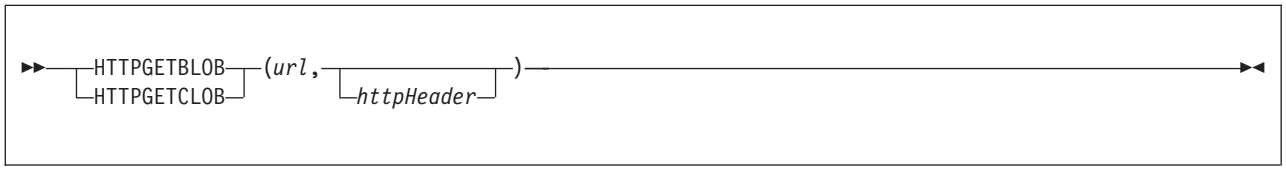
headerAttributeValue
Specifies a value for the *headerAttribute*. Separate *headerAttribute* and *headerAttributeValue* combinations with single spaces.

name
The header name.

value
The header value.

HTTPGETBLOB and HTTPGETCLOB

The HTTPGETBLOB and HTTPGETCLOB REST functions retrieve a binary or text-based resource from the specified URL through an HTTP GET request. HTTPGETBLOB returns the resource as BLOB(5M) data. HTTPGETCLOB returns the resource as CLOB(5M) data. The character set is converted into the database code page if necessary.



The schema is DB2XML.

url
Specifies the URL of the resource being accessed. This argument is defined as a VARCHAR(2048) value.

httpHeader
Specifies an optional header XML document. This argument is defined as a CLOB(10K) value.

The XML header document can provide additional HTTP header values in the following format:

```
<httpHeader headerAttribute="headerAtributeValue">
        <header name="name" value="value" />
</httpHeader>
```

headerAttribute
Specify any of the following optional attributes:

connectionTimeout
Specifies an integer value for the connection timeout threshold in milliseconds.

readTimeout
Specifies an integer value for the read timeout threshold in milliseconds.

followRedirects
Specifies whether redirects should be followed. This is a boolean value.

useCaches
Specifies whether caches should be used. This is a boolean value.

headerAttributeValue
Specifies a value for the *headerAttribute*. Separate *headerAttribute* and *headerAttributeValue* combinations with single spaces.

name
The header name.

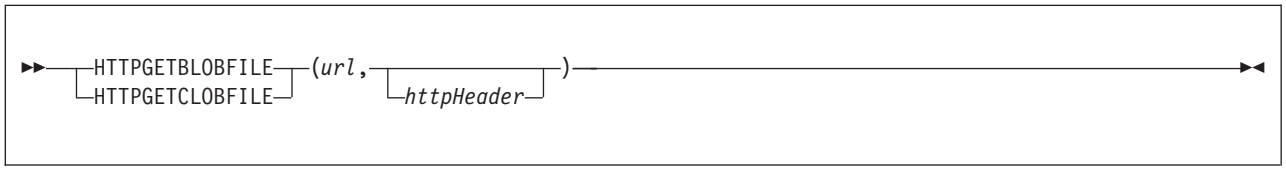
value
The header value.

The following SQL statement retrieves country information from the GeoNames database:

```
SELECT DB2XML.HTTPGETCLOB(
  CAST ('http://ws.geonames.org/countryInfo?lang=' ||
          DB2XML.URLENCODE('en','') ||
        '&country=' ||
          DB2XML.URLENCODE('us','') ||
        '&type=XML' AS VARCHAR(255)),
  CAST(NULL AS CLOB(1K)))
FROM SYSIBM.SYSDUMMY1;
```

HTTPGETBLOBFILE and HTTPGETCLOBFILE

The HTTPGETBLOBFILE and HTTPGETCLOBFILE REST functions retrieve a binary or text-based resource from the specified URL through an HTTP GET request. The resource is stored in a temporary file, and the path of the temporary file is returned as VARCHAR data. The character set is converted into the database code page if necessary.



The schema is DB2XML.

url
Specifies the URL of the resource that is being accessed. This argument is defined as a VARCHAR(2048) value.

httpHeader
Specifies an optional header XML document. This argument is defined as a CLOB(10K) value.

The XML header document can provide additional HTTP header values in the following format:

```
<httpHeader headerAttribute="headerAtributeValue">
        <header name="name" value="value" />
</httpHeader>
```

headerAttribute
Specify any of the following optional attributes:

connectionTimeout
Specifies an integer value for the connection timeout threshold in milliseconds.

readTimeout
Specifies an integer value for the read timeout threshold in milliseconds.

followRedirects
Specifies whether redirects should be followed. This is a boolean value.

useCaches
Specifies whether caches should be used. This is a boolean value.

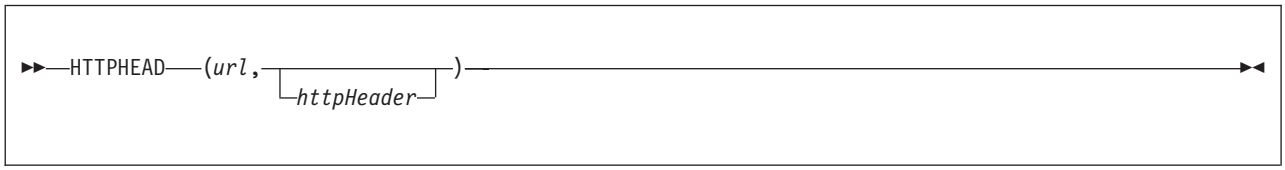
headerAttributeValue
Specifies a value for the *headerAttribute*. Separate *headerAttribute* and *headerAttributeValue* combinations with single spaces.

name
The header name.

value
The header value.

HTTPHEAD

The HTTPHEAD REST function verifies the HTTP header for the specified resource through an HTTP HEAD request. The HTTP header is returned as CLOB or XML data.



The schema is DB2XML.

url
Specifies the URL of the resource. This argument is defined as a VARCHAR(2048) value.

httpHeader
Specifies an optional header XML document. This argument is defined as a CLOB(10K) value.

The XML header document can provide additional HTTP header values in the following format:

```
<httpHeader headerAttribute="headerAtributeValue">
        <header name="name" value="value" />
</httpHeader>
```

headerAttribute
Specify any of the following optional attributes:

connectionTimeout
Specifies an integer value for the connection timeout threshold in milliseconds.

readTimeout
Specifies an integer value for the read timeout threshold in milliseconds.

followRedirects
Specifies whether redirects should be followed. This is a boolean value.

useCaches
Specifies whether caches should be used. This is a boolean value.

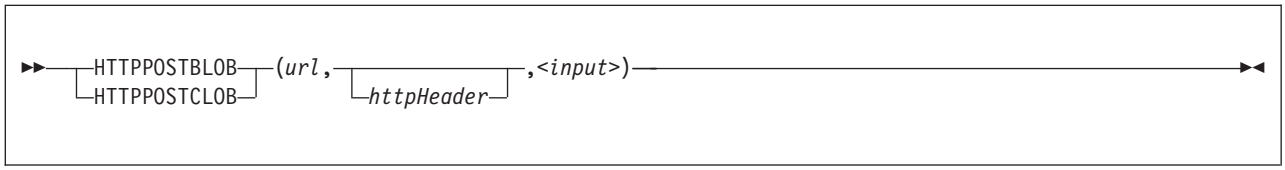
headerAttributeValue
Specifies a value for the *headerAttribute*. Separate *headerAttribute* and *headerAttributeValue* combinations with single spaces.

name
The header name.

value
The header value.

HTTPPOSTBLOB and HTTPPOSTCLOB

The HTTPPOSTBLOB and HTTPPOSTCLOB REST functions update a binary or text-based resource under the specified URL through an HTTP POST request. Response messages from the server are returned as BLOB for HTTPPOSTBLOB or as CLOB for HTTPPOSTCLOB. The character set is converted into the database code page if necessary.



The schema is DB2XML.

url
Specifies the URL at which to update the data. This argument is defined as a VARCHAR(2048) value.

httpHeader
Specifies an optional header XML document. This argument is defined as a CLOB(10K) value.

The XML header document can provide additional HTTP header values in the following format:

```
<httpHeader headerAttribute="headerAtributeValue">
        <header name="name" value="value" />
</httpHeader>
```

headerAttribute
Specify any of the following optional attributes:

connectionTimeout
Specifies an integer value for the connection timeout threshold in milliseconds.

readTimeout
Specifies an integer value for the read timeout threshold in milliseconds.

followRedirects
Specifies whether redirects should be followed. This is a boolean value.

useCaches
Specifies whether caches should be used. This is a boolean value.

headerAttributeValue
Specifies a value for the *headerAttribute*. Separate *headerAttribute* and *headerAttributeValue* combinations with single spaces.

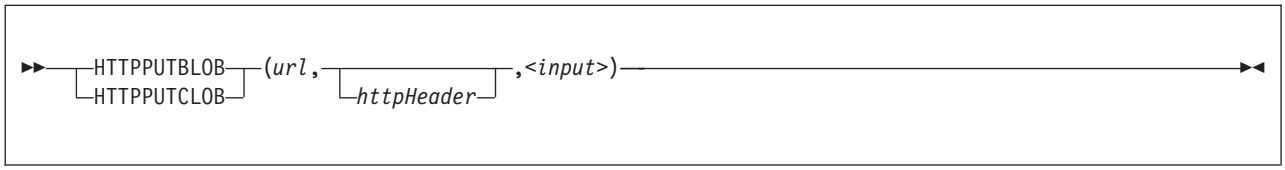
name
The header name.

value
The header value.

<*input***>**
Specifies the data to update at the specified URL. This argument is defined as BLOB(5M) for HTTPPOSTBLOB or CLOB(5M) for HTTPPOSTCLOB.

HTTPPUTBLOB and HTTPPUTCLOB

The HTTPPUTBLOB and HTTPPUTCLOB REST functions create or update a binary or text-based resource under the specified URL through an HTTP PUT request. Response messages from the server are returned as BLOB for HTTPPUTBLOB or as CLOB for HTTPPUTCLOB. The character set is converted into the database code page if necessary.



The schema is DB2XML.

url
Specifies the URL at which to create or update the data. This argument is defined as a VARCHAR(2048) value.

httpHeader
Specifies an optional header XML document. This argument is defined as a CLOB(10K) value.

The XML header document can provide additional HTTP header values in the following format:

```
<httpHeader headerAttribute="headerAtributeValue">
        <header name="name" value="value" />
</httpHeader>
```

headerAttribute
Specify any of the following optional attributes:

connectionTimeout
Specifies an integer value for the connection timeout threshold in milliseconds.

readTimeout
Specifies an integer value for the read timeout threshold in milliseconds.

followRedirects
Specifies whether redirects should be followed. This is a boolean value.

useCaches
Specifies whether caches should be used. This is a boolean value.

headerAttributeValue
Specifies a value for the *headerAttribute*. Separate *headerAttribute* and *headerAttributeValue* combinations with single spaces.

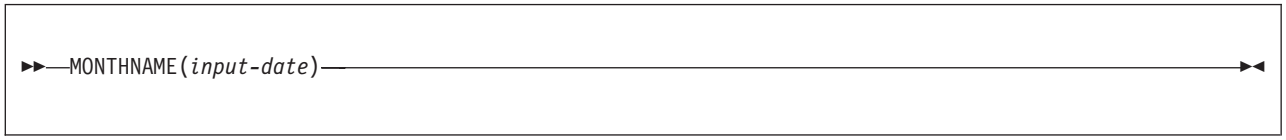
name
The header name.

value
The header value.

<*input***>**
Specifies the data to create or update at the specified URL. This argument is defined as BLOB(5M) for HTTPPUTBLOB or CLOB(5M) for HTTPPUTCLOB.

MONTHNAME

The MONTHNAME function returns the calendar name of the month in which a given date falls. The name is returned in English.



The schema is DSN8.

input-date
A valid date or valid character string representation of a date. A character string representation must have a data type of VARCHAR and an actual length that is no greater than 10 bytes. The date must be in ISO format.

The result of the function is VARCHAR(9).

The MONTHNAME function uses the IBM C++ class IDate.

The following table shows the external and specific names for MONTHNAME. The specific names differ depending on the data type of the input argument.

Table 207. External and specific names for MONTHNAME

Input arguments	External name	Specific name
input-date (VARCHAR)	DSN8EUMN	DSN8.DSN8EUMNV
input-date (DATE)	DSN8EUMN	DSN8.DSN8EUMND

Example 1: For the current date, find the name of the month.

```
VALUES DSN8.MONTHNAME( CURRENT DATE );
```

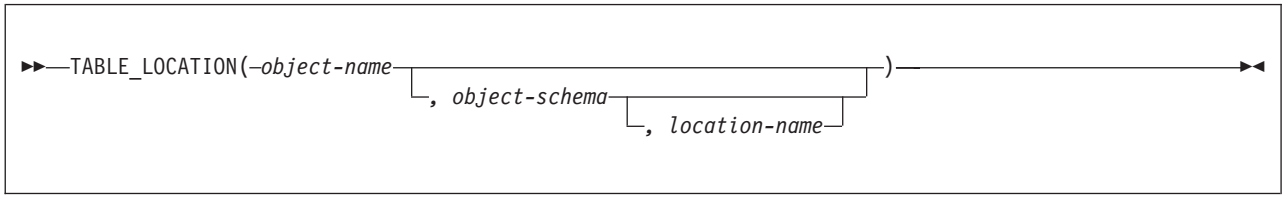
Example 2: Find the month of the year in which Delores Quintana, employee number 000130, was hired.

```
SELECT  FIRSTNME || ' '
    ||  LASTNAME || ' was hired in the month of '
    ||  DSN8.MONTHNAME( HIREDATE )
    ||  CHAR( HIREDATE )
  FROM  EMP
 WHERE  EMPNO  = '000130';
```

The result of the function is 'DELORES QUINTANA was hired in the month of July'.

TABLE_LOCATION

The TABLE_LOCATION function searches for an object and returns the location name of the object after any alias chains have been resolved.



The schema is DSN8.

The starting point of the resolution is the object that is specified by *object-name* and, if specified, *object-schema* and *location-name*. If the starting point does not refer to an alias, the location name of the starting point is returned. The resulting name can be of a table, view, or undefined object. The function returns a blank if there is no location name.

object-name
A character expression that specifies the unqualified name to be resolved. The unqualified name is usually of an existing alias. *object-name* must have a data type of VARCHAR and an actual length that is no greater than 18 bytes.

object-schema
A character expression that represents the schema that is used to qualify the value specified in *object-name* before resolution. *object-schema* must have a data type of VARCHAR and an actual length that is no greater than 8 bytes.

If *object-schema* is not specified or is null, the default schema is used for the qualifier.

location-name
A character expression that represents the location that is used to qualify the value specified in *object-name* before resolution. *location-name* must have a data type of VARCHAR and an actual length that is no greater than 16 bytes.

If *location-name* is not specified or is null, the location name is equivalent to "any".

The result of the function is VARCHAR(16). If *object-name* can be null, the result can be null; if *object-name* is null, the result is the null value.

The following table shows the external and specific names for TABLE_LOCATION. The specific names differ depending on the number of input arguments to the function.

Table 208. External and specific names for TABLE_LOCATION

Input arguments	External name	Specific name
object-name (VARCHAR)	DSN8DUTI	DSN8.DSN8DUTILV
object-name (VARCHAR) *object-schema* (VARCHAR)	DSN8DUTI	DSN8.DSN8DUTILVV

Table 208. External and specific names for TABLE_LOCATION (continued)

Input arguments	External name	Specific name
object-name (VARCHAR) *object-schema* (VARCHAR) *location-name* (VARCHAR)	DSN8DUTI	DSN8.DSN8DUTILVVV

Example: Assume that:

- DSN8.ALIAS_RS_SYSTABLES is an alias of SYSIBM.SYSTABLES at location name 'REMOTE_SITE'.
- The CURRENT SQLID is DSN8.

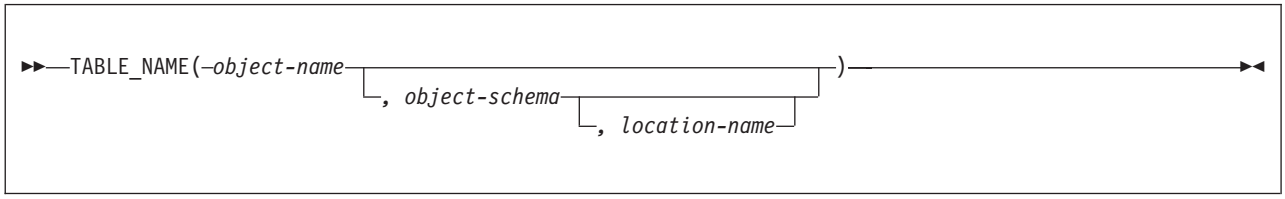
Use TABLE_LOCATION to find the location name where the base object for ALIAS_RS_SYSTABLES resides.

```
VALUES DSN8.TABLE_LOCATION( 'ALIAS_RS_SYSTABLES' );
```

The result of the function is 'REMOTE_SITE'.

TABLE_NAME

The TABLE_NAME function searches for an object and returns the unqualified name of the object after any alias chains have been resolved.



The schema is DSN8.

The starting point of the resolution is the object that is specified by *object-name* and, if specified, *object-schema* and *location name*. If the starting point does not refer to an alias, the unqualified name of the starting point is returned. The resulting name can be of a table, view, or undefined object.

object-name
A character expression that specifies the unqualified name to be resolved. The unqualified name is usually of an existing alias. *object-name* must have a data type of VARCHAR and an actual length that is no greater than 18 bytes.

object-schema
A character expression that represents the schema that is used to qualify the value specified in *object-name* before resolution. *object-schema* must have a data type of VARCHAR and an actual length that is no greater than 8 bytes.

If *object-schema* is not specified or is null, the default schema is used for the qualifier.

location-name
A character expression that represents the location that is used to qualify the value specified in *object-name* before resolution. *location-name* must have a data type of VARCHAR and an actual length than is no greater than 16 bytes.

If *location-name* is not specified or is null, the location name is equivalent to "any".

The result of the function is VARCHAR(128). If *object-name* can be null, the result can be null; if *object-name* is null, the result is the null value.

The following table shows the external and specific names for TABLE_NAME. The specific names differ depending on the number of input arguments to the function.

Table 209. External and specific names for TABLE_NAME

Input arguments	External name	Specific name
object-name (VARCHAR)	DSN8DUTI	DSN8.DSN8DUTINV
object-name (VARCHAR) *object-schema* (VARCHAR)	DSN8DUTI	DSN8.DSN8DUTINVV

Table 209. External and specific names for TABLE_NAME (continued)

Input arguments	External name	Specific name
object-name (VARCHAR) *object-schema* (VARCHAR) *location-name* (VARCHAR)	DSN8DUTI	DSN8.DSN8DUTINVVV

Example: Assume that:

- DSN8.VIEW_OF_SYSTABLES is a view of SYSIBM.SYSTABLES.
- DSN8.ALIAS_OF_VIEW is an alias of DSN8.VIEW_OF_SYSTABLES.
- The CURRENT SQLID is DSN8.

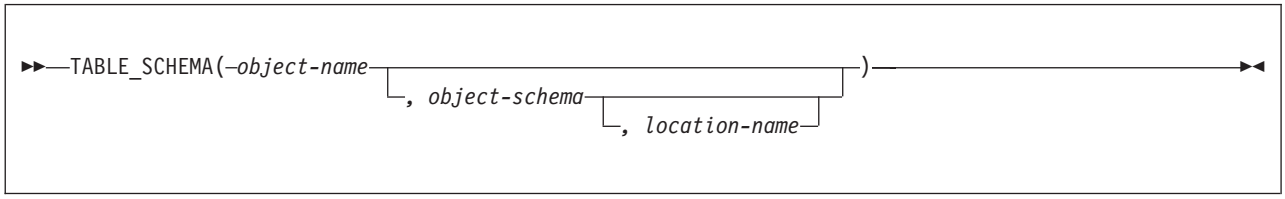
Use TABLE_NAME to find the name of the base object for ALIAS_OF_VIEW.

```
VALUES DSN8.TABLE_NAME( 'ALIAS_OF_VIEW' );
```

The result of the function is 'VIEW_OF_SYSTABLES'.

TABLE_SCHEMA

The TABLE_SCHEMA function searches for an object and returns the schema name of the object after any synonyms or alias chains have been resolved.



The schema is DSN8.

The starting point of the resolution is the object that is specified by *objectname* and *objectschema*. If the starting point does not refer to an alias or synonym, the schema name of the starting point is returned. The resulting schema name can be of a table, view, or undefined object.

object-name
A character expression that specifies the unqualified name to be resolved. The unqualified name is usually of an existing alias. *object-name* must have a data type of VARCHAR and an actual length that is no greater than 18 bytes.

object-schema
A character expression that represents the schema that is used to qualify the value specified in *object-name* before resolution. *object-schema* must have a data type of VARCHAR and an actual length that is no greater than 8 bytes.

If *object-schema* is not specified or is null, the default schema is used for the qualifier.

location-name
A character expression that represents the location that is used to qualify the value specified in *object-name* before resolution. *location-name* must have a data type of VARCHAR (and an actual length that is no greater than 16 bytes.

If *location-name* is not specified or is null, the location name is equivalent to "any".

The result of the function is VARCHAR(128). If *object-name* can be null, the result can be null; if *object-name* is null, the result is the null value.

The following table shows the external and specific names for TABLE_SCHEMA. The specific names differ depending on the number of input arguments.

Table 210. External and specific names for function TABLE_SCHEMA

Input arguments	External name	Specific name
object-name (VARCHAR)	DSN8DUTI	DSN8.DSN8DUTISV
object-name (VARCHAR) *object-schema* (VARCHAR)	DSN8DUTI	DSN8.DSN8DUTISVV

Table 210. External and specific names for function TABLE_SCHEMA (continued)

Input arguments	External name	Specific name
	DSN8DUTI	DSN8.DSN8DUTISVVV
object-name (VARCHAR) *object-schema* (VARCHAR) *location-name* (VARCHAR)		

Example: Assume that:

- DSN8.ALIAS_OF_SYSTABLES is an alias of SYSIBM.SYSTABLES.
- The CURRENT SQLID is DSN8.

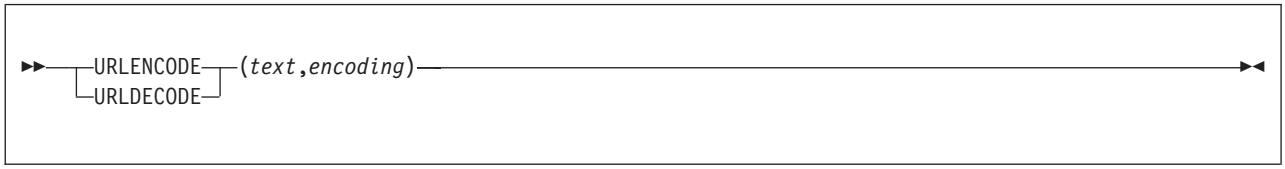
Find the name of the schema of the base table for ALIAS_OF_SYSTABLES.

```
VALUES DSN8.TABLE_SCHEMA( 'ALIAS_OF_SYSTABLES' );
```

The result of the function is 'SYSIBM'.

URLENCODE and URLDECODE

The URLENCODE and URLDECODE helper REST functions complete URL encoding or decoding of the provided text.



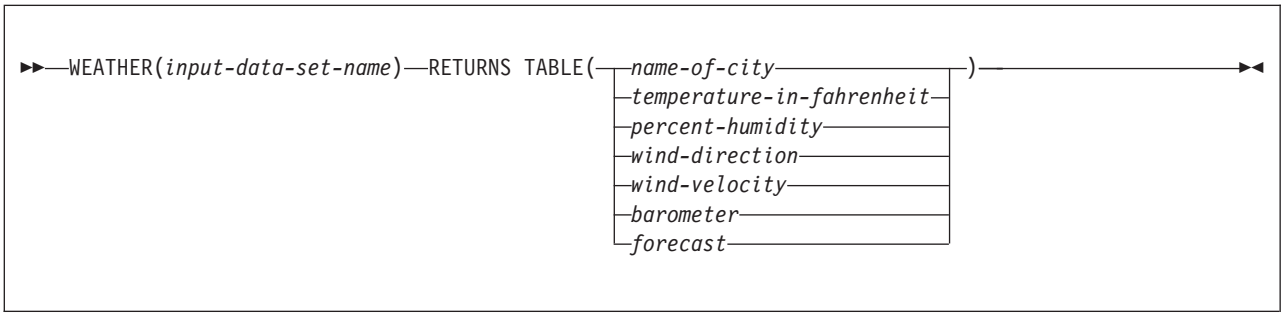
The schema is DB2XML.

text
Specifies the text to encode or decode. This argument is defined as a VARCHAR(2048) value.

encoding
Specifies the character set that is to be used. It can be set to NULL where UTF-8 is used as the default.

WEATHER

The WEATHER function returns information from a TSO data set as a DB2 table. The TSO data set contains sample weather statistics for various cities in the United States. The statistics are returned to the client with a row for each city and a column for each statistic. The WEATHER function is provided primarily to help you design and implement table functions.



The schema is DSN8.

Unlike the other sample user-defined functions, which are scalar functions, WEATHER is a table function. WEATHER shows how to use a table function to make non-relational data available to a client for manipulation by SQL.

input-data-set-name
The name of the TSO data set that contains sample weather statistics. The name is a character string with a data type of VARCHAR and an actual length that is not greater than 44 bytes.

The result of the function is a DB2 table with the following columns. Each column can be null.

name-of-city
VARCHAR(30)

temperature-in-fahrenheit
INTEGER

percent-humidity
INTEGER

wind-direction
VARCHAR(5)

wind-velocity
INTEGER

barometer
FLOAT

forecast
VARCHAR(25)

The external program name for the function is DSN8DUWF, and the specific name is DSN8.DSN8DUWF.

Example: Find the name of and the forecast for the cities that have a temperature less than 25 degrees.

```
SELECT CITY, FORECAST
  FROM TABLE(DSN8.WEATHER('prefix.SDSNIVPD(DSN8LWC)')) AS W
  WHERE TEMP_IN_F < 25
  ORDER BY CITY;
```

This example returns:

```
Bessemer, MI     Slight chance of snow
Cheyenne, WY     Continued cooling
Helena, MT       Heavy snow
Pierre, SD       Continued cold
```

Information resources for DB2 for z/OS and related products

Information about DB2 for z/OS and products that you might use in conjunction with DB2 for z/OS is available in online information centers or on library websites.

Obtaining DB2 for z/OS publications

The current DB2 for z/OS publications are available from the following website:

http://pic.dhe.ibm.com/infocenter/dzichelp/v2r2/topic/com.ibm.db2z11.doc/src/alltoc/db2z_lib.htm

Links to the information center version and the PDF version of each publication are provided.

DB2 for z/OS publications are also available for download from the IBM Publications Center (http://www.ibm.com/shop/publications/order).

In addition, books for DB2 for z/OS are available on a CD-ROM that is included with your product shipment:

- DB2 11 for z/OS Licensed Library Collection, LK5T-8882, in English. The CD-ROM contains the collection of books for DB2 11 for z/OS in PDF format. Periodically, IBM refreshes the books on subsequent editions of this CD-ROM.

Installable information center

You can download or order an installable version of the Information Management Software for z/OS Solutions Information Center, which includes information about DB2 for z/OS, QMF, IMS, and many DB2 and IMS Tools products. You can install this information center on a local system or on an intranet server. For more information, see http://pic.dhe.ibm.com/infocenter/dzichelp/v2r2/topic/com.ibm.dzic.doc/installabledzic.htm.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming interface information

This information is intended to help you to code SQL statements. This information primarily documents General-use Programming Interface and Associated Guidance Information provided by DB2 11 for z/OS. This information also documents Product-sensitive Programming Interface and Associated Guidance Information provided by DB2 11 for z/OS.

General-use Programming Interface and Associated Guidance Information

General-use Programming Interfaces allow the customer to write programs that obtain the services of DB2 11 for z/OS.

Product-sensitive Programming Interface and Associated Guidance Information

Product-sensitive Programming Interfaces allow the customer installation to perform tasks such as diagnosing, modifying, monitoring, repairing, tailoring, or tuning of this IBM software product. Use of such interfaces creates dependencies on the detailed design or implementation of the IBM software product. Product-sensitive Programming Interfaces should be used only for these specialized purposes. Because of their dependencies on detailed design and implementation, it is to be expected that programs written to such interfaces may need to be changed in order to run with new product releases or versions, or as a result of service.

Product-sensitive Programming Interface and Associated Guidance Information is identified where it occurs by the following markings:

PSPI Product-sensitive Programming Interface and Associated Guidance Information... PSPI

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered marks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at http://www.ibm.com/legal/copytrade.shtml.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Privacy policy considerations

IBM Software products, including software as a service solutions, (“Software Offerings”) may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering’s use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM’s Privacy Policy at http://www.ibm.com/privacy and IBM’s Online Privacy Statement at http://www.ibm.com/privacy/details the section entitled “Cookies, Web Beacons and Other Technologies” and the “IBM Software Products and Software-as-a-Service Privacy Statement” at http://www.ibm.com/software/info/product-privacy.

Glossary

The glossary is available in the Information Management Software for z/OS Solutions Information Center.

See the Glossary topic for definitions of DB2 for z/OS terms.

Index

Special characters

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Product Number: 5615-DB2
5697-P43

Printed in USA

SC19-4066-00

Spine information:

DB2 11 for z/OS

SQL Reference