Before using this information and the product it supports, be sure to read the general information under “Notices” on page 261.


This document contains proprietary information of IBM. It is provided under a license agreement and is protected by copyright law. The information contained in this publication does not include any product warranties, and any statements provided in this manual should not be interpreted as such.

You can order IBM publications online or through your local IBM representative.

- To order publications online, go to the IBM Publications Center at http://www.ibm.com/shop/publications/order
- To find your local IBM representative, go to the IBM Directory of Worldwide Contacts at http://www.ibm.com/planetwide

To order publications from IBM in the United States or Canada, call 1-800-IBM-4YOU (426-4968).

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1999, 2004. All rights reserved. US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
GROUP BY clause ................................................................. 56
HAVING clause .................................................................. 57
INNER JOIN ........................................................................ 58
INSERT statement ................................................................. 60
JOIN operation .................................................................... 62
LEFT OUTER JOIN ................................................................. 63
LOCK TABLE statement ......................................................... 64
ORDER BY clause ................................................................. 66
Query .................................................................................. 67
RIGHT OUTER JOIN ................................................................. 69
ScalarSubquery ................................................................... 70
SelectExpression ................................................................. 71
SELECT statement ................................................................. 74
TableExpression .................................................................. 76
TableSubquery .................................................................... 77
UPDATE statement ................................................................. 78
VALUES expression ................................................................. 80
WHERE clause .................................................................... 81
WHERE CURRENT OF clause .................................................... 82
Built-In Functions ................................................................. 83
Standard Built-In Functions ..................................................... 83
Aggregates (Set Functions) ...................................................... 84
ABS or ABSVAL .................................................................... 86
AVG. .................................................................................. 87
BIGINT. ............................................................................... 88
CAST .................................................................................. 89
CHAR .................................................................................. 91
LENGTH ............................................................................. 93
Concatenation .................................................................... 94
NULLIF and CASE expressions. .............................................. 95
COUNT ............................................................................... 96
COUNT(*) .......................................................................... 97
CURRENT_DATE ..................................................................... 98
CURRENT_DATE .................................................................... 99
CURRENT ISOLATION ............................................................. 100
CURRENT_SCHEMA .................................................................. 101
CURRENT_TIME .................................................................... 102
CURRENT_TIME .................................................................... 103
CURRENT_TIMESTAMP .......................................................... 104
CURRENT_TIMESTAMP .......................................................... 105
CURRENT_USER .................................................................... 106
DATE .................................................................................. 107
DAY .................................................................................. 108
DOUBLE. ........................................................................... 109
HOUR. .............................................................................. 110
IDENTITY_VAL_LOCAL .......................................................... 111
INTEGER. .......................................................................... 113
LOCATE ............................................................................. 114
LCASE or LOWER ................................................................. 115
LTRIM. ............................................................................... 116
MAX .................................................................................. 117
MIN .................................................................................. 118
MINUTE. ............................................................................ 119
MOD .................................................................................. 120
MONTH ............................................................................... 121
RTRIM .............................................................................. 122
SECOND. ........................................................................... 123
SESSION_USER .................................................................... 124
SMALLINT .......................................................................... 125
SQRT ................................................................................. 126
SUBSTR .............................................................................. 127
SUM ................................................................. 128
TIME ............................................................... 129
TIMESTAMP ......................................................... 130
UCASE or UPPER ...................................................... 131
USER ................................................................. 132
VARCHAR ........................................................... 133
YEAR ................................................................. 134
Built-in system functions ........................................... 135
SYSCS_UTIL.SYSCS_CHECK_TABLE ......................... 135
SYSCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS ............. 135
SYSCS_UTIL.SYSCS_GET_DATABASEPROPERTY ............... 135
Built-in system procedures ....................................... 136
SYSCS_UTIL.SYSCS_COMPRESS_TABLE ....................... 136
SYSCS_UTIL.SYSCS_SET_RUNTIMESTATISTICS ............... 137
SYSCS_UTIL.SYSCS_SET_STATISTICS_TIMING ............... 138
SYSCS_UTIL.SYSCS_SET_DATABASEPROPERTY ............... 138
SYSCS_UTIL.SYSCS_FREEZE_DATABASE ....................... 139
SYSCS_UTIL.SYSCS_UNFREEZE_DATABASE ..................... 139
SYSCS_UTIL.SYSCS_CHECKPOINT_DATABASE .................. 140
SYSCS_UTIL.SYSCS_BACKUP_DATABASE ....................... 140
SYSCS_UTIL.SYSCS_EXPORT_TABLE ......................... 140
SYSCS_UTIL.SYSCS_EXPORT_QUERY ......................... 141
SYSCS_UTIL.SYSCS_IMPORT_TABLE ......................... 142
SYSCS_UTIL.SYSCS_IMPORT_DATA .............................. 143
Data Types .......................................................... 145
Built-In Type Overview ............................................ 145
Numeric Types ....................................................... 146
  Numeric Type Overview ........................................ 146
  Numeric Type Promotion in Expressions .................... 146
  Storing Values of One Numeric Data Type in Columns of Another Numeric Data Type .............................. 147
  Scale for Decimal Arithmetic ................................ 147
Data type assignments and comparison, sorting, and ordering ............................................. 149
BIGINT ............................................................... 153
BLOB ................................................................. 154
CHAR ................................................................. 155
CHAR FOR BIT DATA ............................................... 156
CLOB ................................................................. 157
DATE ................................................................. 159
DECIMAL .......................................................... 160
DOUBLE ........................................................... 161
DOUBLE PRECISION ............................................... 162
FLOAT ............................................................... 163
INTEGER ............................................................ 164
LONG VARCHAR ..................................................... 165
LONG VARCHAR FOR BIT DATA .................................... 166
NUMERIC ........................................................... 167
REAL ................................................................. 168
SMALLINT .......................................................... 169
TIME ............................................................... 170
TIMESTAMP ........................................................ 171
VARCHAR ........................................................... 172
VARCHAR FOR BIT DATA ............................................ 173
SQL Expressions .................................................... 174
Expression Precedence ............................................ 176
  Example .......................................................... 177
Boolean expression ............................................... 177
Dynamic Parameters ............................................... 179
  Dynamic Parameters Example .................................. 179
  Where Dynamic Parameters Are Allowed ..................... 179
Chapter 2. SQL Reserved Words ................................. 183
### Chapter 7. Database Connection URL Attributes

- `bootPassword=key` .......................................... 234
- `create=true` ....................................................... 235
- `databaseName=nameofDatabase` ................................. 236
- `dataEncryption=true` ........................................... 237
- `encryptionProvider=providerName` ............................ 238
- `encryptionAlgorithm=algorithm` ............................... 239
- `territory=CC` ....................................................... 240
- `logDevice=logDirectoryPath` ................................ 241
- `password=password` ............................................. 242
- `rollForwardRecoveryFrom=Path` .............................. 243
- `createFrom=Path` ............................................... 244
- `restoreFrom=Path` ............................................... 245
- `shutdown=true` .................................................... 246
- `user=user` ......................................................... 247
- (no attributes) .................................................. 248

### Chapter 8. J2EE Compliance: Java Transaction API and java.sql Extensions

- J2EE Overview ..................................................... 249
  - JVM and Libraries for J2EE Features ......................... 250
  - The JTA API ..................................................... 250
    - Notes on Product Behavior ................................ 250
    - Recovered Global Transactions ................................ 250
    - XAConnections and User Names and Passwords .................. 250
  - java.sql: JDBC Extensions ................................... 251

### Appendix A. Cloudscape API

- Stand-Alone Tools and Utilities ................................ 253
- JDBC Implementation Classes ................................... 253
Appendix B. Supported territories ................................................................. 255
Appendix C. Cloudscape limitations .............................................................. 257
  Identifier length limitations ................................................................. 257
  Numeric limitations ............................................................................. 257
  String limitations ............................................................................... 258
  DATE, TIME, and TIMESTAMP limitations ...................................... 258
  Database Manager limitations .............................................................. 259

Notices ........................................................................................................ 261
  Trademarks ........................................................................................ 263

Index .......................................................................................................... 265
About This Document

- “Purpose of This Document” on page ix
- “Audience” on page ix
- “How This Document Is Organized” on page ix

For general information about the Cloudscape™ documentation, such as a complete list of books, conventions, and further reading, see *Getting Started with IBM Cloudscape*.

Purpose of This Document

This book, the *IBM Cloudscape Reference Manual*, provides reference information about Cloudscape. It covers Cloudscape’s SQL language, the Cloudscape implementation of JDBC, Cloudscape system catalogs, Cloudscape error messages, Cloudscape properties, and SQL keywords.

Audience

This book is a reference for Cloudscape users, typically application developers. Cloudscape users who are not familiar with the SQL standard or the Java programming language will benefit from consulting books on those topics.

Cloudscape users who want a how-to approach to working with Cloudscape or an introduction to Cloudscape concepts should read the *IBM Cloudscape Developer’s Guide*.

How This Document Is Organized

This document includes the following chapters:

- **Chapter 1, “SQL Language Reference,”** on page 7
  Reference information about Cloudscape’s SQL language, including manual pages for statements, functions, and other syntax elements.

- **Chapter 2, “SQL Reserved Words,”** on page 183
  SQL keywords beyond the standard SQL-92 keywords.

- **Chapter 3, “Cloudscape Support for SQL-92 Features,”** on page 187
  A list of SQL-92 features that Cloudscape does and does not support.

- **Chapter 4, “Cloudscape System Tables,”** on page 193
  Reference information about the Cloudscape system catalogs.

- **Chapter 5, “Cloudscape Exception Messages and SQL States,”** on page 201
  Information about Cloudscape exception messages.

- **Chapter 6, “JDBC Reference,”** on page 203
  Information about Cloudscape’s implementation of the JDBC interface include support for JDBC 2.0 features.

- **Chapter 7, “Database Connection URL Attributes,”** on page 233
  Information about the supported attributes to Cloudscape’s JDBC database connection URL.

- **Chapter 8, “J2EE Compliance: Java Transaction API and javax.sql Extensions,”** on page 249
Information about the supported attributes to Cloudscape’s support for the Java Transaction API.

- Appendix A, “Cloudscape API,” on page 253

Notes about proprietary APIs for Cloudscape.
Chapter 1. SQL Language Reference

Cloudscape Version 10 implements an SQL-92 core subset, as well as some SQL-99 features.

This document provides an overview of the current SQL language by describing the statements, built-in functions, data types, expressions, and special characters it contains. It includes the following sections:

- “Capitalization and Special Characters”
- “SQL Identifiers” on page 2
- “Statements” on page 6
- “Built-In Functions” on page 83
- “Data Types” on page 145
- “SQL Expressions” on page 174

Capitalization and Special Characters

Using the classes and methods of JDBC, you submit SQL statements to Cloudscape as strings. The character set permitted for strings containing SQL statements is Unicode. Within these strings, the following rules apply:

- Double quotation marks delimit special identifiers referred to in SQL-92 as delimited identifiers.
- Single quotation marks delimit character strings.
- Within a character string, to represent a single quotation mark or apostrophe, use two single quotation marks. (In other words, a single quotation mark is the escape character for a single quotation mark.)
  A double quotation mark does not need an escape character. To represent a double quotation mark, simply use a double quotation mark. However, note that in a Java program, a double quotation mark requires the backslash escape character.
  -- a single quotation mark is the escape character
  -- for a single quotation mark

VALUES 'Joe''s umbrella'
-- in ij, you don't need to escape the double quotation marks
VALUES 'He said, "hello!"

n = stmt.executeUpdate(
    "UPDATE aTable setStringcol = 'He said, \"hello!\"';");

- SQL keywords are case-insensitive. For example, you can type the keyword SELECT as SELECT, Select, select, or sELECT.
- SQL-92-style identifiers are case-insensitive (see SQL92Identifier), unless they are delimited.
- Java-style identifiers are always case-sensitive.
- * is a wildcard within a SelectExpression. See “The * Wildcard” on page 72. It can also be the multiplication operator. In all other cases, it is a syntactical metasymbol that flags items you can repeat 0 or more times.
SQL Identifiers

An identifier is the representation within the language of items created by the user, as opposed to language keywords or commands. Some identifiers stand for dictionary objects, which are the objects you create—such as tables, views, indexes, columns, and constraints—that are stored in a database. They are called dictionary objects because Cloudscape stores information about them in the system tables, sometimes known as a data dictionary. SQL also defines ways to alias these objects within certain statements.

Each kind of identifier must conform to a different set of rules. Identifiers representing dictionary objects must conform to SQL-92 identifier rules and are thus called SQL92Identifiers.

Rules for SQL92Identifiers

Ordinary identifiers are identifiers not surrounded by double quotation marks. Delimited identifiers are identifiers surrounded by double quotation marks.

An ordinary identifier must begin with a letter and contain only letters, underscore characters (_), and digits. The permitted letters and digits include all Unicode letters and digits, but Cloudscape does not attempt to ensure that the characters in identifiers are valid in the database’s locale.

A delimited identifier can contain any characters within the double quotation marks. The enclosing double quotation marks are not part of the identifier; they serve only to mark its beginning and end. Spaces at the end of a delimited identifier are insignificant (truncated). Cloudscape translates two consecutive double quotation marks within a delimited identifier as one double quotation mark—that is, the “translated” double quotation mark becomes a character in the delimited identifier.

Periods within delimited identifiers are not separators but are part of the identifier (the name of the dictionary object being represented).

So, in the following example:

"A.B"

is a dictionary object, while

"A"."B"

is a dictionary object qualified by another dictionary object (such as a column named “B” within the table “A”).

SQL92Identifier

An SQL92Identifier is a dictionary object identifier that conforms to the rules of SQL-92. SQL-92 states that identifiers for dictionary objects are limited to 128 characters and are case-insensitive (unless delimited by double quotes), because
they are automatically translated into uppercase by the system. You cannot use reserved words as identifiers for dictionary objects unless they are delimited. If you attempt to use a name longer than 128 characters, SQLException X0X11 is raised.

Cloudscape defines keywords beyond those specified by the SQL-92 standard (see Chapter 2, “SQL Reserved Words,” on page 183).

Example:
-- the view name is stored in the
-- system catalogs as ANIDENTIFIER
CREATE VIEW AnIdentifier (RECEIVED) AS VALUES 1
-- the view name is stored in the system
-- catalogs with case intact
CREATE VIEW "ACaseSensitiveIdentifier" (RECEIVED) AS VALUES 1

This section describes the rules for using SQL92Identifiers to represent the following dictionary objects:
- schema–Name
- table–Name
- correlation–Name
- column–Name
- Simple–column–Name
- index–Name
- constraint–Name
- cursor–Name
- TriggerName
- AuthorizationIdentifier

Qualifying Dictionary Objects
Since some dictionary objects can be contained within other objects, you can qualify those dictionary object names. Each component is separated from the next by a period. An SQL92Identifier is “dot-separated.” You qualify a dictionary object name in order to avoid ambiguity.

schema–Name
A schema–Name represents a schema. Schemas contain other dictionary objects, such as tables and indexes. Schemas provide a way to name a subset of tables and other dictionary objects within a database.

You can explicitly create or drop a schema. The default user schema is the APP schema (if no user name is specified at connection time). You are not allowed to create dictionary objects in the SYS schema.

Thus, you can qualify references to tables with the schema name. When a schema–Name is not specified, the default schema name is implicitly inserted. System tables are placed in the SYS schema. You must qualify all references to system tables with the SYS schema identifier. For more information about system tables, see Chapter 4, “Cloudscape System Tables,” on page 193.

A schema is hierarchically the highest level of dictionary object, so you cannot qualify a schema–Name.

Syntax:
Example:
-- SAMP.EMPLOYEE is a table–Name qualified by a schema–Name
SELECT COUNT(*) FROM SAMP.EMPLOYEE
-- You must qualify system catalog names with their schema, SYS
SELECT COUNT(*) FROM SYS.SysColumns

**table–Name**

A table–Name represents a table. You can qualify a table–Name with a schema–Name.

Syntax:

```
[ [schema–Name] ] SQL92Identifier
```

Example:

-- SAMP.PROJECT is a table–Name that includes a schema–Name
SELECT COUNT(*) FROM SAMP.PROJECT

**new–table–Name**

A new–table–Name represents a renamed table. You cannot qualify a new–table–Name with a schema–Name.

Syntax:

```
SQL92Identifier
```

Example:

-- FlightBooks is a new–table–Name that does not include a schema–Name
RENAME TABLE FLIGHTAVAILABILITY TO FLIGHTAVAILABLE

**view–Name**

A view–Name represents a table or a view. You can qualify a view–Name with a schema–Name.

Syntax:

```
[ [schema–Name] ] SQL92Identifier
```

Example:

-- This is a View qualified by a schema–Name
SELECT COUNT(*) FROM SAMP.EMP_RESUME

**correlation–Name**

A correlation–Name is given to a table expression in a FROM clause as a new name or alias for that table. You do not qualify a correlation–Name with a schema–Name.

Syntax:

```
SQL92Identifier
```

Example:

-- C is a correlation–Name
SELECT C.NAME FROM SAMP.STAFF C
column–Name

In many places in the SQL syntax, you can represent the name of a column by qualifying it with a `table–Name` or `correlation–Name`.

In some situations, you cannot qualify a `column–Name` with a `table–Name` or a `correlation–Name`, but must use a `Simple–column–Name` instead. Those situations are:

- creating a `CREATE TABLE` statement
- specifying updatable columns in a cursor
- in a column’s correlation name in a `SELECT` expression (see `SelectExpression`)
- in a column’s correlation name in a `TableExpression` (see `TableExpression`)

Syntax:

\[
[\{table–Name | correlation–Name\}.] SQL92Identifier
\]

Example:

```sql
-- C.Country is a column–Name qualified with a correlation–Name
SELECT C.Country
FROM APP.Countries C
```

Simple–column–Name

A `Simple–column–Name` is used to represent a column when it cannot be qualified by a `table–Name` or `correlation–Name`. This is the case when the qualification is fixed, as it is in a column definition within a `CREATE TABLE` statement, and in `ORDER BY` clauses.

Syntax:

```sql
SQL92Identifier
```

Example:

```sql
-- country is a Simple–column–Name
CREATE TABLE CONTINENT (COUNTRY VARCHAR(26) NOT NULL PRIMARY KEY, COUNTRY_ISO_CODE CHAR(2), REGION VARCHAR(26))
```

index–Name

An `index–Name` represents an index. Indexes live in schemas, so you can qualify their names with `schema–Names`. Indexes on system tables are in the SYS schema.

Syntax:

```
[Schema–Name]. SQL92Identifier
```

Example:

```sql
DROP INDEX APP.ORIGINDEX;
-- OrigIndex is an index–Name without a schema–Name
CREATE INDEX ORIGINDEX ON FLIGHTS (ORIG_AIRPORT)
```

constraint–Name

You cannot qualify constraint-names.

Syntax:

```sql
SQL92Identifier
```

Example:
CREATE TABLE DETAILED_MAPS (COUNTRY_ISO_CODE CHAR(2)
CONSTRAINT country_fk2 REFERENCES COUNTRIES)

**cursor–Name**

A cursor–Name refers to a cursor. No SQL language command exists to assign a name to a cursor. Instead, you use the JDBC API to assign names to cursors or to retrieve system-generated names. For more information, see the IBM Cloudscape Developer’s Guide. If you assign a name to a cursor, you can refer to that name from within SQL statements.

You cannot qualify a cursor–Name.

**Syntax:**

```sql
SQL92Identifier
```

**Example:**

```java
stmt.executeUpdate("UPDATE SAMP.STAFF SET COMM = "
"COMM + 20 " + "WHERE CURRENT OF " + ResultSet.getCursorName());
```

**TriggerName**

A TriggerName refers to a trigger created by a user.

**Syntax:**

```sql
[Schema–Name.] SQL92Identifier
```

**Example:**

```sql
DROP TRIGGER TRIG1
```

**AuthorizationIdentifier**

User names within the Cloudscape system are known as authorization identifiers. The authorization identifier represents the name of the user, if one has been provided in the connection request. The default schema for a user is equal to its authorization identifier. User names can be case-sensitive within the authentication system, but they are always case-insensitive within Cloudscape’s authorization system unless they are delimited. For more information, see the IBM Cloudscape Developer’s Guide.

**Syntax:**

```sql
SQL92Identifier
```

**Example:**

```java
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.fullAccessUsers', 'Amber,FRED')
```

**Statements**

This section provides manual pages for both high-level language constructs and parts thereof. For example, the CREATE INDEX statement is a high-level statement that you can execute directly via the JDBC interface. This section also includes clauses, which are not high-level statements and which you cannot execute directly but only as part of a high-level statement. The ORDER BY and WHERE clauses are examples of this kind of clause. Finally, this section also includes some
syntactically complex portions of statements called expressions, for example `SelectExpression` and `TableSubquery`. These clauses and expressions receive their own manual pages for ease of reference.

Unless it is explicitly stated otherwise, you can execute or prepare and then execute all the high-level statements, which are all marked with the word `statement`, via the interfaces provided by JDBC. This manual indicates whether an expression can be executed as a high-level statement.

The sections below provide general information about statement use, and descriptions of the specific statements.

- “Interaction with the Dependency System”
- “ALTER TABLE statement” on page 8
- “CREATE Statements” on page 13
- “DROP Statements” on page 28
- “RENAME Statements” on page 35
- “SET Statements” on page 38

Interaction with the Dependency System

Cloudscape internally tracks the dependencies of prepared statements, which are SQL statements that are precompiled before being executed. Typically they are prepared (precompiled) once and executed multiple times.

Prepared statements depend on the dictionary objects and statements they reference. (Dictionary objects include tables, columns, constraints, indexes, views, and triggers.) Removing or modifying the dictionary objects or statements on which they depend invalidates them internally, which means that Cloudscape will automatically try to recompile the statement when you execute it. If the statement fails to recompile, the execution request fails. However, if you take some action to restore the broken dependency (such as restoring the missing table), you can execute the same prepared statement, because Cloudscape will recompile it automatically at the next execute request.

Statements depend on one another—an UPDATE WHERE CURRENT statement depends on the statement it references. Removing the statement on which it depends invalidates the UPDATE WHERE CURRENT statement.

In addition, prepared statements prevent execution of certain DDL statements if there are open results sets on them.

Manual pages for each statement detail what actions would invalidate that statement, if prepared.

Here is an example using the Cloudscape tool `ij`:

```sql
ij> CREATE TABLE mytable (mycol INT);
0 rows inserted/updated/deleted
ij> INSERT INTO mytable VALUES (1), (2), (3);
3 rows inserted/updated/deleted
ij> -- this example uses the ij command prepare,
   -- which prepares a statement
   prepare pl AS 'INSERT INTO MyTable VALUES (4)';
ij> -- pl depends on mytable;
execute pl;
1 row inserted/updated/deleted
ij> -- Cloudscape executes it without recompiling
CREATE INDEX i1 ON mytable(mycol);
```
0 rows inserted/updated/deleted
ij> -- pl is temporarily invalidated because of new index
execute pl;
1 row inserted/updated/deleted
ij> -- Cloudscape automatically recompiles pl and executes it
DROP TABLE mytable;
0 rows inserted/updated/deleted
ij> -- Cloudscape permits you to drop table
-- because result set of pl is closed
-- however, the statement pl is temporarily invalidated
CREATE TABLE mytable (mycol INT);
0 rows inserted/updated/deleted
ij> INSERT INTO mytable VALUES (1), (2), (3);
3 rows inserted/updated/deleted
ij> execute pl;
1 row inserted/updated/deleted
ij> -- Because pl is invalid, Cloudscape tries to recompile it
-- before executing.
-- It is successful and executes.
DROP TABLE mytable;
0 rows inserted/updated/deleted
ij> -- statement pl is now invalid,
-- and this time the attempt to recompile it
-- upon execution will fail
execute pl;
ERROR 42X05: Table 'MYTABLE' does not exist.

ALTER TABLE statement
ALTER TABLE

The ALTER TABLE statement allows you to:

- add a column to a table
- add a constraint to a table
- drop an existing constraint from a table
- increase the width of a VARCHAR, CHAR VARYING, and CHARACTER VARYING column
- add a default value for an existing column in a table
- override row-level locking for the table (or drop the override)

Format:

```sql
ALTER TABLE table-name
{
  ADD COLUMN column-definition |
  ADD CONSTRAINT clause |
  DROP { PRIMARY KEY | FOREIGN KEY constraint-name | UNIQUE constraint-name | CHECK constraint-name | CONSTRAINT constraint-name }
  ALTER column-alteration |
  LOCKSIZE { ROW | TABLE }
}
```

`column-definition`:

```sql
Simple-column-Name  DataType
[Column-Level-Constraint]*
[ [ WITH ] DEFAULT {ConstantExpression | NULL } ]
```

`column-alteration`:

```sql
column-Name  SET DATA TYPE VARCHAR(integer) |
column-name  SET INCREMENT BY integer-constant
```

In the `column-alteration`, `SET INCREMENT BY` integer-constant, specifies the interval between consecutive values of the identity column. The next value to be generated for the identity column will be determined from the last assigned value with the increment applied. The column must already be defined with the IDENTITY attribute.

ALTER TABLE does not affect any view that references the table being altered. This includes views that have an “*” in their SELECT list. You must drop and re-create those views if you wish them to return the new columns.

The modifications you can make using ALTER TABLE are explained in the following sections:

- “Adding Columns”
- “Adding Constraints” on page 10
- “Dropping Constraints” on page 10
- “Modifying Columns” on page 10
- “Setting Defaults” on page 11
- “Changing the Lock Granularity for the Table” on page 11

Adding Columns: The syntax for the `column-definition` for a new column is the same as for a column in a CREATE TABLE statement. This means that a column constraint can be placed on the new column within the ALTER TABLE ADD COLUMN statement. However, a column with a NOT NULL constraint can be
added to an existing table only if the table is empty or if you give a default value; otherwise, an exception is thrown when the ALTER TABLE statement is executed.

Just as in CREATE TABLE, if the column definition includes a unique or primary key constraint, the column cannot contain null values, so the NOT NULL attribute must also be specified (SQLSTATE 42831).

See “Adding Constraints” for the other limitations.

Note: If a table has an UPDATE trigger without an explicit column list, adding a column to that table in effect adds that column to the implicit update column list upon which the trigger is defined, and all references to transition variables are invalidated so that they pick up the new column.

Adding Constraints: ALTER TABLE ADD CONSTRAINT adds a table-level constraint to an existing table. Any supported table-level constraint type can be added via ALTER TABLE. The following limitations exist on adding a constraint to an existing table:

• When adding a foreign key or check constraint to an existing table, Cloudscape checks the table to make sure existing rows satisfy the constraint. If any row is invalid, Cloudscape throws a statement exception and the constraint is not added.

• All columns included in a primary key must contain non null data and be unique.

ALTER TABLE ADD UNIQUE or PRIMARY KEY provide a shorthand method of defining a primary key composed of a single column. If PRIMARY KEY is specified in the definition of column C, the effect is the same as if the PRIMARY KEY(C) clause were specified as a separate clause. The column cannot contain null values, so the NOT NULL attribute must also be specified.

For information on the syntax of constraints, see “CONSTRAINT clause” on page 43. Use the syntax for table-level constraint when adding a constraint with the ADD TABLE ADD CONSTRAINT syntax.

Dropping Constraints: ALTER TABLE DROP CONSTRAINT drops a constraint on an existing table. To drop an unnamed constraint, you must specify the generated constraint name stored in SYS.SYSCONSTRAINTS as a delimited identifier.

Dropping a primary key, unique, or foreign key constraint drops the physical index that enforces the constraint (also known as a backing index).

Modifying Columns: The column-alteration allows you to alter the named column in the following ways:

• Increasing the length of an existing VARCHAR column. CHARACTER VARYING or CHAR VARYING can be used as synonyms for the VARCHAR keyword.

    To increase the width of a column of these types, specify the data type and new size after the column name.

    You are not allowed to decrease the width or to change the data type. You are not allowed to increase the width of a column that is part of a primary or unique key referenced by a foreign key constraint or that is part of a foreign key constraint.

• Specifying the interval between consecutive values of the identity column.
• To set an interval between consecutive values of the identity column, specify the integer-constant.
• You must previously define the column with the IDENTITY attribute (SQLSTATE 42837).

**Setting Defaults:** You can specify a default value for a new column or add a default value to an existing column (see “Modifying Columns” on page 10). A default value is the value that is inserted into a column if no other value is specified. If not explicitly specified, the default value of a column is NULL. If you add a default to a new column, existing rows in the table gain the default value in the new column. If you add a default to an existing column, existing rows in the table do not gain the default value in the new column.

You can set the interval between consecutive values of the identity column to an existing column in a table. If there are existing rows in the table, the values in the column for which the SET INCREMENT default was added do not change. Note that this means that values in the column are not guaranteed to be unique (use a unique or primary key constraint to guarantee uniqueness).

For more information about defaults, see “CREATE TABLE statement” on page 20.

**Changing the Lock Granularity for the Table:** The LOCKSIZE clause allows you to override row-level locking for the specific table, if your system uses the default setting of row-level locking. (If your system is set for table-level locking, you cannot change the locking granularity to row-level locking, although Cloudscape allows you to use the LOCKSIZE clause in such a situation without throwing an exception.) To override row-level locking for the specific table, set locking for the table to TABLE. If you created the table with table-level locking granularity, you can change locking back to ROW with the LOCKSIZE clause in the ALTER TABLE statement. For information about why this is sometimes useful, see “About the System’s Selection of Lock Granularity” in Chapter 4 of *Tuning IBM Cloudscape*.

**Examples:**

-- Add a new column with a column-level constraint
-- to an existing table
-- An exception will be thrown if the table
-- contains any rows
-- since the newcol will be initialized to NULL
-- in all existing rows in the table
**ALTER TABLE CITIES ADD COLUMN REGION VARCHAR(26) CONSTRAINT NEW_CONSTRAINT CHECK (REGION IS NOT NULL);**

-- Add a new unique constraint to an existing table
-- An exception will be thrown if duplicate keys are found
**ALTER TABLE SAMP.DEPARTMENT ADD CONSTRAINT NEW_UNIQUE UNIQUE (DEPTNO);**

-- add a new foreign key constraint to the
-- Cities table. Each row in Cities is checked
-- to make sure it satisfied the constraints.
-- if any rows don't satisfy the constraint, the
-- constraint is not added
**ALTER TABLE CITIES ADD CONSTRAINT COUNTRY_FK FOREIGN KEY (COUNTRY) REFERENCES COUNTRIES (COUNTRY);**

-- Add a primary key constraint to a table
-- First, create a new table
**CREATE TABLE ACTIVITIES (CITY_ID INT NOT NULL, SEASON CHAR(2), ACTIVITY VARCHAR(32) NOT NULL);**
-- You will not be able to add this constraint if the
-- columns you are including in the primary key have
-- null data or duplicate values.
ALTER TABLE Activities ADD PRIMARY KEY (city_id, activity);

-- Drop a primary key constraint from the CITIES table
ALTER TABLE Cities DROP CONSTRAINT Cities_PK;
-- Drop a foreign key constraint from the CITIES table
ALTER TABLE Cities DROP CONSTRAINT COUNTRIES_FK;
-- add a DEPTNO column with a default value of 1
ALTER TABLE SAMP.EMP_ACT ADD COLUMN DEPTNO INT DEFAULT 1;
-- increase the width of a VARCHAR column
ALTER TABLE SAMP.EMP_PHOTO ALTER PHOTO_FORMAT SET DATA TYPE VARCHAR(30);
-- change the lock granularity of a table
ALTER TABLE SAMP.SALES LOCKSIZE TABLE;

Results: An ALTER TABLE statement causes all statements that are dependent on
the table being altered to be recompiled before their next execution. ALTER TABLE
is not allowed if there are any open cursors that reference the table being altered.
CREATE Statements

- “CREATE FUNCTION Statement”
- “CREATE INDEX statement” on page 15
- “CREATE PROCEDURE Statement” on page 17
- “CREATE SCHEMA statement” on page 19
- “CREATE TABLE statement” on page 20
- “CREATE TRIGGER statement” on page 23
- “CREATE VIEW statement” on page 27

CREATE FUNCTION Statement

The CREATE FUNCTION statement allows you to create Java functions, which you can then use in an expression.

Syntax:
CREATE FUNCTION [schema–Name. ] function–Name ( [FunctionParameter [, FunctionParameter] ] * )
RETURNS DataType [FunctionElement ] *

function–Name:
[ schema–Name ] SQL92Identifier

If schema–Name is not provided, the current schema is the default schema. If a qualified procedure name is specified, the schema name cannot begin with SYS.

FunctionParameter:
[ parameter–Name ] DataType

ParameterName must be unique within a function.

The syntax of DataType is described in “Data Types” on page 145.

Note: Long data–types such as LONG VARCHAR, LONG VARCHAR FOR BIT DATA, CLOB, and BLOB are not allowed as parameters in a CREATE FUNCTION statement.

FunctionElement:
{
SPECIFIC [ schema–Name ] SQL92Identifier
| LANGUAGE { JAVA }
| EXTERNAL NAME string
| PARAMETER STYLE JAVA
| { NO SQL | CONTAINS SQL | READES SQL DATA }
| { RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT }
}

LANGUAGE:

JAVA- the database manager will call the function as a public static method in a Java class.

EXTERNAL NAME string:

String describes the Java method to be called when the function is executed, and takes the following form:

class_name.method_name
The External Name cannot have any extraneous spaces.

PARAMETER STYLE:

JAVA - The function will use a parameter-passing convention that conforms to the Java language and SQL Routines specification. INOUT and OUT parameters will be passed as single entry arrays to facilitate returning values. Result sets are returned through additional parameters to the Java method of type java.sql.ResultSet[].

Cloudscape does not support long column types (for example Long Varchar, BLOB, and so on). An error will occur if you try to use one of these long column types.

NO SQL, CONTAINS SQL, READS SQL DATA:

Indicates whether the function issues any SQL statements and, if so, what type.

CONTAINS SQL
    Indicates that 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
    Indicates that the function cannot execute any SQL statements

READS SQL DATA
    Indicates that some SQL statements that do not modify SQL data can be included in the function. Statements that are not supported in any stored function return a different error. This is the default value.

RETURNS NULL ON NULL INPUT or CALLED ON NULL INPUT:

Specifies whether the function is called if any of the input arguments is null. The result is the null value.

RETURNS NULL ON NULL INPUT
    Specifies that the function is not invoked if any of the input arguments is null. The result is the null value.

CALLED ON NULL INPUT
    Specifies that the function is invoked if any or all input arguments are null. This specification means that the function must be coded to test for null argument values. The function can return a null or non-null value. This is the default setting.

The function elements may appear in any order, but each type of element can only appear once. A function definition must contain these elements:

• LANGUAGE
• PARAMETER STYLE
• EXTERNAL NAME

Example:
CREATE FUNCTION TO_DEGREES(RADIANS DOUBLE) RETURNS DOUBLE
PARAMETER STYLE JAVA NO SQL LANGUAGE JAVA
EXTERNAL NAME 'java.lang.Math.toDegrees'
CREATE INDEX statement

A CREATE INDEX statement creates an index on a table. Indexes can be on one or more columns in the table.

Syntax:
```
CREATE [UNIQUE] INDEX index-Name
ON table-Name (Simple-column-Name [ ASC | DESC ]
, Simple-column-Name [ ASC | DESC ])*
```

In addition, while there is no maximum number of columns for an index key in Cloudscape. The maximum number of columns for an index key is 16.

An index name cannot exceed 18 characters.

A column must not be named more than once in a single CREATE INDEX statement. Different indexes can name the same column, however.

Cloudscape can use indexes to improve the performance of data manipulation statements (see Tuning IBM Cloudscape). In addition, UNIQUE indexes provide a form of data integrity checking.

Index names are unique within a schema. (Some database systems allow different tables in a single schema to have indexes of the same name, but Cloudscape does not.) Both index and table are assumed to be in the same schema if a schema name is specified for one of the names, but not the other. If schema names are specified for both index and table, an exception will be thrown if the schema names are not the same. If no schema name is specified for either table or index, the current schema is used.

By default, Cloudscape uses the ascending order of each column to create the index. Specifying ASC after the column name does not alter the default behavior. The DESC keyword after the column name causes Cloudscape to use descending order for the column to create the index. Using the descending order for a column can help improve the performance of queries that require the results in mixed sort order or descending order and for queries that select the minimum or maximum value of an indexed column.

If a qualified index name is specified, the schema name cannot begin with SYS.

Indexes and Constraints: Unique, primary key, and foreign key constraints generate indexes that enforce or “back” the constraint (and are thus sometimes called backing indexes). If a column or set of columns has a UNIQUE or PRIMARY KEY constraint on it, you can not create an index on those columns. Cloudscape has already created it for you with a system-generated name. System-generated names for indexes that back up constraints are easy to find by querying the system tables if you name your constraint. For example, to find out the name of the index that backs a constraint called FLIGHTS_PK:

```
SELECT CONGLOMERATENAME FROM SYS.SYSCONGLOMERATES,
SYS.SYSCONSTRAINTS WHERE
SYS.SYSCONGLOMERATES.TABLEID = SYSCONSTRAINTS.TABLEID
AND CONSTRAINTNAME = 'FLIGHTS_PK'
```

Examples:
```
CREATE INDEX OrigIndex ON Flights(orig_airport);
-- money is usually ordered from greatest to least,
-- so create the index using the descending order
```
CREATE INDEX PAY_DESC ON SAMP.EMPLOYEE (SALARY);
-- use a larger page size for the index
call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY('derby.storage.pageSize','8192');
CREATE INDEX IXSALE ON SAMP.SALES (SALES);
call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY('derby.storage.pageSize',NULL);

Page Size and Key Size:

**Note:** The size of the key columns in an index must be equal to or smaller than half the page size. If the length of the key columns in an existing row in a table is larger than half the page size of the index, creating an index on those key columns for the table will fail. This error only occurs when creating an index if an existing row in the table fails the criteria. After an index is created, inserts may fail if the size of their associated key exceeds the criteria.

**Statement Dependency System:** Prepared statements that involve SELECT, INSERT, UPDATE, UPDATE WHERE CURRENT, DELETE, and DELETE WHERE CURRENT on the table referenced by the CREATE INDEX statement are invalidated when the index is created. Open cursors on the table are not affected.
CREATE PROCEDURE Statement
The CREATE PROCEDURE statement allows you to create Java stored procedures, which you can then call using the CALL PROCEDURE statement.

Syntax:
CREATE PROCEDURE procedure–Name ( [ ProcedureParameter [, ProcedureParameter ] ] * )
[ ProcedureElement ] *

procedure–Name:
[ Schema–Name ] SQL92Identifier

If schema–Name is not provided, the current schema is the default schema. If a qualified procedure name is specified, the schema name cannot begin with SYS.

ProcedureParameter:
[ { IN | OUT | INOUT } ] [ parameter–Name ] DataType

The default value for a parameter is IN. ParameterName must be unique within a procedure.

The syntax of DataType is described in “Data Types” on page 145.

Note: Long data–types such as LONG VARCHAR, LONG VARCHAR FOR BIT DATA, CLOB, and BLOB are not allowed as parameters in a CREATE PROCEDURE statement.

ProcedureElement:
{ SPECIFIC [ Schema–Name ] SQL92Identifier
| { DYNAMIC | RESULT SETS } INTEGER
| LANGUAGE { JAVA } 
| EXTERNAL NAME string 
| PARAMETER STYLE JAVA
| { NO SQL | MODIFIES SQL DATA | CONTAINS SQL | READS SQL DATA } }

DYNAMIC RESULT SETS integer:
Indicates the estimated upper bound of returned result sets for the procedure. Default is no (zero) dynamic result sets.

LANGUAGE:
JAVA- the database manager will call the procedure as a public static method in a Java class.

EXTERNAL NAME string:
String describes the Java method to be called when the procedure is executed, and takes the following form:
class_name.method_name

The External Name cannot have any extraneous spaces.

PARAMETER STYLE:
JAVA - The procedure will use a parameter-passing convention that conforms to the Java language and SQL Routines specification. INOUT and OUT parameters will be passed as single entry arrays to facilitate returning values. Result sets are returned through additional parameters to the Java method of type java.sql.ResultSet[] that are passed single entry arrays.

Cloudscape does not support long column types (for example Long Varchar, BLOB, and so on). An error will occur if you try to use one of these long column types.

NO SQL, CONTAINS SQL, READS SQL DATA, MODIFIES SQL DATA:

Indicates whether the stored procedure issues any SQL statements and, if so, what type.

CONTAINS SQL
Indicates that SQL statements that neither read nor modify SQL data can be executed by the stored procedure. Statements that are not supported in any stored procedure return a different error. MODIFIES SQL DATA is the default value.

NO SQL
Indicates that the stored procedure cannot execute any SQL statements

READS SQL DATA
Indicates that some SQL statements that do not modify SQL data can be included in the stored procedure. Statements that are not supported in any stored procedure return a different error.

MODIFIES SQL DATA
Indicates that the stored procedure can execute any SQL statement except statements that are not supported in stored procedures.

The procedure elements may appear in any order, but each type of element can only appear once. A procedure definition must contain these elements:

- LANGUAGE
- PARAMETER STYLE
- EXTERNAL NAME

Example:

```
CREATE PROCEDURE SALES.TOTAL_REVENUE(IN S_MONTH INTEGER,
IN S_YEAR INTEGER, OUT TOTAL DECIMAL(10,2))
PARAMETER STYLE JAVA READS SQL DATA LANGUAGE JAVA EXTERNAL NAME
'com.acme.sales.calculateRevenueByMonth'
```
CREATE SCHEMA statement

A schema is a way to logically group objects in a single collection and provide a unique namespace for objects.

Syntax:

```sql
CREATE SCHEMA Schema-Name
```

The CREATE SCHEMA statement is used to create a schema. A schema name cannot exceed 30 characters. Schema names must be unique within the database.

Examples:

-- Create a schema for employee-related tables
CREATE SCHEMA EMP;

-- Create a schema for airline-related tables
CREATE SCHEMA Flights

-- Create a table called "Availability" in each schema
CREATE TABLE FLIGHTS.AVAILABILITY
(FLIGHT_ID CHAR(6) NOT NULL, SEGMENT_NUMBER INT NOT NULL, FLIGHT_DATE DATE NOT NULL, ECONOMY_SEATS_TAKEN INT, BUSINESS_SEATS_TAKEN INT, FIRSTCLASS_SEATS_TAKEN INT,
CONSTRAINT FLT_AVAIL_PK PRIMARY KEY (FLIGHT_ID, SEGMENT_NUMBER, FLIGHT_DATE));

CREATE TABLE EMP.AVAILABILITY
(HOTEL_ID INT NOT NULL, BOOKING_DATE DATE NOT NULL, ROOMS_TAKEN INT,
CONSTRAINT HOTELAVAIL_PK PRIMARY KEY (HOTEL_ID, BOOKING_DATE));
CREATE TABLE statement

A CREATE TABLE statement creates a table. Tables contain columns and constraints, rules to which data must conform. Table-level constraints specify a column or columns. Columns have a data type and can specify column constraints (column-level constraints).

For information about constraints, see "CONSTRAINT clause" on page 43.

You can specify a default value for a column. A default value is the value to be inserted into a column if no other value is specified. If not explicitly specified, the default value of a column is NULL. See "Column Default."

You can specify storage properties such as page size for a table by calling the SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY system procedure.

If a qualified table name is specified, the schema name cannot begin with SYS.

Syntax:
CREATE TABLE table-Name
( [column-definition | Table-Level Constraint] [, column-definition | Table-Level Constraint] ) *

column-definition:
Simple-column-Name DataType
[ Column-Level-Constraint ]*

[ Column-Level-Constraint ]*
generated-column-spec:
[ [ WITH ] DEFAULT {ConstantExpression | NULL } ]
[ GENERATED ALWAYS AS IDENTITY
[ ( START WITH IntegerConstant ] ]
[ ,INCREMENT BY IntegerConstant] ] ]

The syntax of Data-Type is described in "Data Types" on page 145.

The syntaxes of Column-Level-Constraint and Table-Level Constraint are described in "CONSTRAINT clause" on page 43.

The IDENTITY keyword can only be specified if the data type associated with the column is one of the following exact integer types.

- SMALLINT
- INT
- BIGINT

Column Default: For the definition of a default value, a ConstantExpression is an expression that does not refer to any table. It can include constants, date-time special registers, current schemas, users, and null.

Identity column attributes: For SMALLINT, INT, and BIGINT columns with identity attributes, Cloudscape automatically assigns increasing integer values to the column. Identity column attributes behave like other defaults in that when an insert statement does not specify a value for the column, Cloudscape automatically provides the value. However, the value is not a constant; Cloudscape automatically
increments the default value every time a row is inserted. Also, unlike other defaults, you are not allowed to insert a value directly into or update an identity column.

By default, the initial value of an identity column is 1, and the amount of the increment is 1. You can specify non-default values for both the initial value and the interval amount when you define the column with the key words START WITH and INCREMENT BY. And if you specify a negative number for the increment value, Cloudscape decrements the value with each insert. If this value is 0, or positive, Cloudscape increments the value with each insert.

The maximum and minimum values allowed in identity columns are determined by the data type of the column. Attempting to insert a value outside the range of values supported by the data type raises an exception.

### Table 1. Maximum and Minimum Values for Columns with Generated Column Specs

<table>
<thead>
<tr>
<th>Data type</th>
<th>Maximum Value</th>
<th>Minimum Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>32767 (java.lang.Short.MAX_VALUE)</td>
<td>-32768 (java.lang.Short.MIN_VALUE)</td>
</tr>
<tr>
<td>INT</td>
<td>2147483647 (java.lang.Integer.MAX_VALUE)</td>
<td>-2147483648 (java.lang.Integer.MIN_VALUE)</td>
</tr>
<tr>
<td>BIGINT</td>
<td>9223372036854775807 (java.lang.Long.MAX_VALUE)</td>
<td>-9223372036854775808 (java.lang.Long.MIN_VALUE)</td>
</tr>
</tbody>
</table>

Automatically generated values in an identity column are unique. Use a primary key or unique constraint on a column to guarantee uniqueness. Creating an identity column does *not* create an index on the column.

The IDENTIFICATION function is a non-deterministic function that returns the most recently assigned value for an identity column. See "IDENTITY VALUES" on page 111 for more information.

**Note:** Specify the schema, table, and column name using the same case as those names are stored in the system tables—that is, all upper case unless you used delimited identifiers when creating those database objects.

Cloudsage keeps track of the last increment value for a column in a cache. It also stores the value of what the next increment value will be for the column on disk in the AUTOINCREMENT column of theSYS.SYSCOLUMNS system table. Rolling back a transaction does not undo this value, and thus rolled-back transactions can leave “gaps” in the values automatically inserted into an identity column. Cloudsage behaves this way to avoid locking a row in SYS.SYSCOLUMNS for the duration of a transaction and keeping concurrency high.

When an insert happens within a triggered-SQL-statement, the value inserted by the triggered-SQL-statement into the identity column is available from ConnectionInfo only within the trigger code. The trigger code is also able to see the value inserted by the statement that caused the trigger to fire. However, the statement that caused the trigger to fire is not able to see the value inserted by the triggered-SQL-statement into the identity column. Likewise, triggers can be nested (or recursive). An SQL statement can cause trigger T1 to fire. T1 in turn executes an SQL statement that causes trigger T2 to fire. If both T1 and T2 insert rows into a table that cause Cloudscape to insert into an identity column, trigger T1 cannot see the value caused by T2’s insert, but T2 can see the value caused by T1’s insert. Each nesting level can see increment values generated by itself and previous
nesting levels, all the way to the top-level SQL statement that initiated the recursive triggers. You can only have 16 levels of trigger recursion.

Examples:

```
CREATE TABLE HOTELAVAILABILITY
  (HOTEL_ID INT NOT NULL, BOOKING_DATE DATE NOT NULL,
   ROOMS_TAKEN INT DEFAULT 0, PRIMARY KEY (HOTEL_ID, BOOKING_DATE));
-- the table-level primary key definition allows you to
-- include two columns in the primary key definition
PRIMAR Y KEY (hotel_id, booking_date))
-- assign an identity column attribute to an INTEGER
-- column, and also define a primary key constraint
-- on the column
CREATE TABLE PEOPLE
  (PERSON_ID INT NOT NULL GENERATED ALWAYS AS IDENTITY
   CONSTRAINT PEOPLE_PK PRIMARY KEY,
   PERSON VARCHAR(26));
-- assign an identity column attribute to a SMALLINT
-- column with an initial value of 5 and an increment value
-- of 5.
CREATE TABLE GROUPS
  (GROUP_ID SMALLINT NOT NULL GENERATED ALWAYS AS IDENTITY
   (START WITH 5, INCREMENT BY 5),
   ADDRESS VARCHAR(100),
   PHONE VARCHAR(15));
```

Note: For more examples of CREATE TABLE statements using the various constraints, see “CONSTRAINT clause” on page 43.
CREATE TRIGGER statement

A trigger defines a set of actions that are executed when a database event occurs on a specified table. A database event is a delete, insert, or update operation. For example, if you define a trigger for a delete on a particular table, the trigger’s action occurs whenever someone deletes a row or rows from the table.

Along with constraints, triggers can help enforce data integrity rules with actions such as cascading deletes or updates. Triggers can also perform a variety of functions such as issuing alerts, updating other tables, sending e-mail, and other useful actions.

You can define any number of triggers for a single table, including multiple triggers on the same table for the same event.

You can create a trigger in any schema except SYS. The trigger need not reside in the same schema as the table on which it is defined.

If a qualified trigger name is specified, the schema name cannot begin with SYS.

Syntax:

```
CREATE TRIGGER TriggerName
AFTER { INSERT | DELETE | UPDATE [ OF column-Name [, column-Name] ]
ON table-Name
[ ReferencingClause ]
FOR EACH { ROW | STATEMENT }
MODE DB2SQL
Triggered-SQL-statement
```

ReferencingClause:

```
REFERENCING
{
{ OLD | NEW } [ AS ] correlation-Name [{ OLD | NEW } [ AS ] correlation-Name ] |
{ OLD_TABLE | NEW_TABLE } [ AS ] Identifier [{ OLD_TABLE | NEW_TABLE } |
[AS] Identifier ]
}
```

Before or After: When Triggers Fire: Triggers fire after all constraints have been satisfied and after the changes have been applied to the target table. Also called After triggers, they can be either row or statement triggers (see “Statement versus Row Triggers” on page 24).

Insert, Delete, or Update: What Causes the Trigger to Fire: A trigger is fired by one of the following database events, depending on how you define it (in Syntax above, see the third line):

- INSERT
- UPDATE
- DELETE

You can define any number of triggers for a given event on a given table. For update, you can specify columns.

Referencing Old and New Values: The Referencing Clause: Many triggered-SQL-statements need to refer to data that is currently being changed by the database event that caused them to fire. The triggered-SQL-statement might need to refer to the new (post-change or “after”) values.
Cloudscape provides you with a number of ways to refer to data that is currently being changed by the database event that caused the trigger to fire. The easiest way to refer to the changed data in the triggered–SQL–statement is use the transition variables or transition tables.

The referencing clause allows you to provide a correlation name or alias for these transition variables by specifying OLD/NEW AS correlation–Name.

For example, if you add the following clause to the trigger definition:

```
REFERENCING OLD AS DELETEDROW
```

you can then refer to this correlation name in the triggered–SQL–statement:

```
DELETE FROM HotelAvailability WHERE hotel_id = DELETEDROW.hotel_id
```

The OLD and NEW transition variables map to a java.sql.ResultSet with a single row.

**Note:** Only row triggers (see “Statement versus Row Triggers”) can use the transition variables. INSERT row triggers cannot reference an OLD row. DELETE row triggers cannot reference a NEW row.

For statement triggers, transition tables serve as a table identifier for the triggered–SQL–statement or the trigger qualification. The referencing clause allows you to provide a correlation name or alias for these transition tables by specifying OLD_TABLE/NEW_TABLE AS correlation-Name.

For example:

```
REFERENCING OLD_TABLE AS DeletedHotels
```

allows you to use that new identifier (DeletedHotels) in the triggered–SQL–statement:

```
DELETE FROM HotelAvailability WHERE hotel_id IN (SELECT hotel_id FROM DeletedHotels)
```

The old and new transition tables map to a java.sql.ResultSet with cardinality equivalent to the number of rows affected by the triggering event.

**Note:** Only statement triggers (see “Statement versus Row Triggers”) can use the transition tables. INSERT statement triggers cannot reference an OLD table. DELETE statement triggers cannot reference a NEW table.

The referencing clause can designate only one new correlation or identifier and only one old correlation or identifier. Row triggers cannot designate an identifier for a transition table and statement triggers cannot designate a correlation for transition variables.

**Statement versus Row Triggers:** You must specify whether a trigger is a statement trigger or a row trigger:

- **statement triggers**
  A statement trigger fires once per triggering event and regardless of whether any rows are modified by the insert, update, or delete event.

- **row triggers**
  A row trigger fires once for each row affected by the triggering event. If no rows are affected, the trigger does not fire.
Note: An update that sets a column value to the value that it originally contained (for example, UPDATE T SET C = C) causes a row trigger to fire, even though the value of the column is the same as it was prior to the triggering event.

Triggered–SQL–statement: The action defined by the trigger is called the triggered–SQL–statement (in Syntax above, see the last line). It has the following limitations:

- It must not contain any dynamic parameters (?)
- It must not create, alter, or drop the table upon which the trigger is defined.
- It must not add an index to or remove an index from the table on which the trigger is defined.
- It must not add a trigger to or drop a trigger from the table upon which the trigger is defined.
- It must not commit or roll back the current transaction or change the isolation level.
- It must not execute a CALL statement.

The triggered–SQL–statement can reference database objects other than the table upon which the trigger is declared. If any of these database objects is dropped, the trigger is invalidated. If the trigger cannot be successfully recompiled upon the next execution, the invocation throws an exception and the statement that caused it to fire will be rolled back.

For more information on triggered–SQL–statements, see the IBM Cloudscape Developer’s Guide.

Order of Execution: When a database event occurs that fires a trigger, Cloudscape performs actions in this order:

- It performs constraint checking (primary key, unique key, foreign key, check).
- It performs the insert, update, or delete.
- It fires after triggers.

When multiple triggers are defined for the same database event for the same table for the same trigger time (before or after), triggers are fired in the order in which they were created.

Examples:

```sql
-- Statements and after triggers:
CREATE TRIGGER FLIGHTSDELETE
AFTER DELETE ON FLIGHTS
REFERENCING OLD_TABLE AS DELETEDFLIGHTS
FOR EACH STATEMENT MODE DB2SQL
DELETE FROM FLIGHTAVAILABILITY WHERE FLIGHT_ID IN
(SELECT FLIGHT_ID FROM DELETEDFLIGHTS);

CREATE TRIGGER FLIGHTSDELETE3
AFTER DELETE ON FLIGHTS
REFERENCING OLD AS OLD
FOR EACH ROW MODE DB2SQL
DELETE FROM FLIGHTAVAILABILITY WHERE FLIGHT_ID = OLD.FLIGHT_ID;
```

Note: You can find more examples in the IBM Cloudscape Developer’s Guide.

Trigger Recursion:
The maximum trigger recursion depth is 16.

**Related Information:** Special system functions that return information about the current time or current user are evaluated when the trigger fires, not when it is created. Such functions include:

- `CURRENT_DATE`
- `CURRENT_TIME`
- `CURRENT_TIMESTAMP`
- `CURRENT_USER`
- `SESSION_USER`
- `USER`
CREATE VIEW statement

Views are virtual tables formed by a query. A view is a dictionary object that you can use until you drop it.

Views are not updatable.

If a qualified view name is specified, the schema name cannot begin with SYS.

Syntax:
CREATE VIEW [view-Name]
 [ ( Simple-column-Name [ , Simple-column-Name ] * ) ]
 AS Query

A view definition can contain an optional view column list to explicitly name the columns in the view. If there is no column list, the view inherits the column names from the underlying query. All columns in a view must be uniquely named.

Examples:
CREATE VIEW SAMP.V1 (COL_SUM, COL_DIFF)
 AS SELECT COMM + BONUS, COMM - BONUS
 FROM SAMP.EMPLOYEE;

CREATE VIEW SAMP.VEMP_RES (RESUME)
 AS VALUES 'Delores M. Quintana', 'Heather A. Nicholls', 'Bruce Adamson';

CREATE VIEW SAMP.PROJ_COMBO
 (PROJNO, PRENDATE, PRSTAFF, MAJPROJ)
 AS SELECT PROJNO, PRENDATE, PRSTAFF, MAJPROJ
 FROM SAMP.PROJECT UNION ALL
 SELECT PROJNO, EMSTDATE, EMPTIME, EMPNO
 FROM SAMP.EMP_ACT
 WHERE EMPNO IS NOT NULL;

Statement Dependency System: View definitions are dependent on the tables and views referenced within the view definition. DML (data manipulation language) statements that contain view references depend on those views, as well as the objects in the view definitions that the views are dependent on. Statements that reference the view depend on indexes the view uses; which index a view uses can change from statement to statement based on how the query is optimized. For example, given:
CREATE TABLE T1 (C1 DOUBLE PRECISION);

CREATE FUNCTION SIN (DATA DOUBLE)
 RETURNS DOUBLE EXTERNAL NAME 'java.lang.Math.sin'
 LANGUAGE JAVA PARAMETER STYLE JAVA;

CREATE VIEW V1 (C1) AS SELECT SIN(C1) FROM T1;

the following SELECT:
SELECT * FROM V1

is dependent on view V1, table T1, and external scalar function SIN.
DROP Statements

- "DROP FUNCTION"
- "DROP INDEX statement" on page 29
- "DROP PROCEDURE" on page 30
- "DROP SCHEMA statement" on page 31
- "DROP TABLE statement" on page 32
- "DROP TRIGGER statement" on page 33
- "DROP VIEW statement" on page 34

DROP FUNCTION

Syntax:

DROP FUNCTION function-name

Identifies the particular function to be dropped, and is valid only if there is exactly one function instance with the function-name in the schema. The identified function can have any number of parameters defined for it. If no function with the indicated name in the named or implied schema, an error (SQLSTATE 42704) will occur. An error will also occur if there is more than one specific instance of the function in the named or implied schema.
DROP INDEX statement

DROP INDEX removes the specified index.

Syntax:
```sql
DROP INDEX [Index-Name]
```

Examples:
```
DROP INDEX OrigIndex
DROP INDEX DestIndex
```

Statement Dependency System: If there is an open cursor on the table from which the index is dropped, the DROP INDEX statement generates an error and does not drop the index. Otherwise, statements that depend on the index’s table are invalidated.
DROP PROCEDURE

Syntax:

```
DROP PROCEDURE procedure-name
```

Identifies the particular procedure to be dropped, and is valid only if there is exactly one procedure instance with the `procedure-name` in the schema. The identified procedure can have any number of parameters defined for it. If no procedure with the indicated name in the named or implied schema, an error (SQLSTATE 42704) will occur. An error will also occur if there is more than one specific instance of the procedure in the named or implied schema.
**DROP SCHEMA statement**

The DROP SCHEMA statement drops a schema. The target schema must be empty for the drop to succeed.

Neither the *APP* schema (the default user schema) nor the *SYS* schema can be dropped.

**Syntax:**

```
DROP SCHEMA Schema-Name RESTRICT
```

The RESTRICT keyword enforces the rule that no objects can be defined in the specified schema for the schema to be deleted from the database. The RESTRICT keyword is required.

**Examples:**

-- Drop the SAMP schema
-- The SAMP schema may only be deleted from the database
-- if no objects are defined in the SAMP schema.

```
DROP SCHEMA SAMP RESTRICT
```
DROP TABLE statement

DROP TABLE removes the specified table.

Syntax:
DROP TABLE table-Name

Statement Dependency System: Triggers, constraints (primary, unique, check and references from the table being dropped) and indexes on the table are silently dropped. The existence of an open cursor that references table being dropped cause the DROP TABLE statement to generate an error, and the table is not dropped.

Dropping a table invalidates statements that depend on the table. (Invalidating a statement causes it to be recompiled upon the next execution. See "Interaction with the Dependency System" on page 7.)
**DROP TRIGGER statement**

DROP TRIGGER removes the specified trigger.

**Syntax:**
```
DROP TRIGGER [TriggerName]
```

**Examples:**
```
DROP TRIGGER TRIG1
```

**Statement Dependency System:** When a table is dropped, all triggers on that table are automatically dropped. (You don’t have to drop a table’s triggers before dropping the table.)
DROP VIEW statement

Drops the specified view.

Syntax:
DROP VIEW view-Name

Examples:
DROP VIEW AnIdentifier

Statement Dependency System: Any statements referencing the view are invalidated on a DROP VIEW statement. DROP VIEW is disallowed if there are any views or open cursors dependent on the view. The view must be dropped before any objects that it is dependent on can be dropped.
RENAME Statements

- “RENAME INDEX statement” on page 36
- “RENAME TABLE statement” on page 37
**RENAME INDEX statement**

This statement allows you to rename an index in the current schema. Users cannot rename indexes in the SYS schema.

**Syntax:**

```
RENAME INDEX index-Name TO new-index-Name
```

**Examples:**

```
RENAME INDEX DESTINDEX TO ARRIVALINDEX
```

**Statement Dependency System:** RENAME INDEX is not allowed if there are any open cursors that reference the index being renamed.
**RENAME TABLE statement**

RENAME TABLE allows you to rename an existing table in any schema (except the schema SYS).

**Syntax:**

```
RENAME TABLE table-Name TO new-table-Name
```

If there is a view or foreign key that references the table, attempts to rename it will generate an error. In addition, if there are any check constraints or triggers on the table, attempts to rename it will also generate an error.

**Examples:**

```
RENAME TABLE samp.EMP_ACT TO EMPLOYEE_ACT
```

Also see “ALTER TABLE” on page 9 for more information.

**Statement Dependency System:** If there is an index defined on the table, the table can be renamed.

RENAME TABLE is not allowed if there are any open cursors that reference the table being altered.
SET Statements

- “SET SCHEMA statement” on page 39
- “SET CURRENT ISOLATION statement” on page 41
SET SCHEMA statement

The SET SCHEMA statement sets the default schema for a connection’s session to the designated schema. The default schema is used as the target schema for all statements issued from the connection that do not explicitly specify a schema name.

The target schema must exist for the SET SCHEMA statement to succeed. If the schema doesn’t exist an error is returned. See “CREATE SCHEMA statement” on page 19.

The SET SCHEMA statement is not transactional: If the SET SCHEMA statement is part of a transaction that is rolled back, the schema change remains in effect.

Syntax:

```sql
SET [CURRENT] SCHEMA [=]
{
  [schema–Name] | USER | ? | '<string-constant>'
}

| SET CURRENT SQLID [=]
{
  [schema–Name] | USER | ? | '<string-constant>'
}
```

`schema–Name` is an identifier with a maximum length of 128. It is case insensitive unless enclosed in double quotes. (For example, SYS is equivalent to sYs, SYs, sys, etcetera.)

USER is the current user. If no current user is defined, the current schema defaults the APP schema. (If a user name was specified upon connection, the user’s name is the default schema for the connection, if a schema with that name exists.)

? is a dynamic parameter specification that can be used in prepared statements. The SET SCHEMA statement can be prepared once and then executed with different schema values. The schema values are treated as string constants so they are case sensitive. For example, to designate the APP schema, use the string “APP” rather than “app”.

Examples:

-- the following are all equivalent and will work
-- assuming a schema called HOTEL
SET SCHEMA HOTEL
SET SCHEMA hotel
SET CURRENT SCHEMA hotel
SET CURRENT SQLID hotel
SET SCHEMA = hotel
SET CURRENT SCHEMA = hotel
SET CURRENT SQLID = hotel
SET SCHEMA "HOTEL" -- quoted identifier
SET SCHEMA 'HOTEL' -- quoted string
  --This example produces an error because
  --lower case hotel won't be found
SET SCHEMA = 'hotel'
  --This example produces an error because SQLID is not
  --allowed without CURRENT
SET SQLID hotel

Chapter 1. SQL Language Reference   39
-- This sets the schema to the current user id
SET CURRENT SCHEMA USER
// Here's an example of using set schema in a Java program
PreparedStatement ps = conn.prepareStatement("set schema ?");
ps.setString(1,"HOTEL");
ps.executeUpdate();
... do some work
ps.setString(1,"APP");
ps.executeUpdate();

ps.setString(1,"app"); // error - string is case sensitive
// no app will be found
ps.setNull(1, Types.VARCHAR); // error - null is not allowed
SET CURRENT ISOLATION statement

The SET CURRENT ISOLATION LEVEL statement allows a user to change the isolation level for the user’s connection. Valid levels are SERIALIZABLE, REPEATABLE READ, READ COMMITTED, and READ UNCOMMITTED.

Issuing this command commits the current transaction, which is consistent with the java.sql.Connection.setTransactionLevel method.

For information about isolation levels, see “Locking, Concurrency, and Isolation” in the IBM Cloudscape Developer’s Guide.

Syntax:

```
SET [ CURRENT ] ISOLATION [ = ]
{ UR | DIRTY READ | READ UNCOMMITTED
  CS | READ COMMITTED | CURSOR STABILITY
  RS
  RR | REPEATABLE READ | SERIALIZABLE
  RESET
}
```

Examples:

```
set isolation serializable;
```
CALL (PROCEDURE)

The CALL (PROCEDURE) statement is used to call procedures. A call to a procedure does not return any value.

Syntax:
CALL "procedure–Name" ( ? [ , ?]* )

Example:
CREATE PROCEDURE SALES.TOTAL_REVENUE(IN S_MONTH INTEGER,
IN S_YEAR INTEGER, OUT TOTAL DECIMAL(10,2))
PARAMETER STYLE JAVA READS SQL DATA LANGUAGE JAVA EXTERNAL NAME
'com.acme.sales.calculateRevenueByMonth';
CALL SALES.TOTAL_REVENUE(? ,? ,?);
A CONSTRAINT clause is an optional part of a CREATE TABLE statement or ALTER TABLE. A constraint is a rule to which data must conform. Constraint names are optional.

A CONSTRAINT can be one of the following:

- a column-level constraint
  Column-level constraints refer to a single column in the table and do not specify a column name (except check constraints). They refer to the column that they follow.

- a table-level constraint
  Table-level constraints refer to one or more columns in the table. Table-level constraints specify the names of the columns to which they apply. Table-level CHECK constraints can refer to 0 or more columns in the table.

Column constraints include:

- NOT NULL
  Specifies that this column cannot hold NULL values (constraints of this type are not nameable).

- NULL
  The opposite of NOT NULL (not really a constraint), it specifies that the column can hold NULL values. Specifying NULL is the same as saying nothing at all, except when the column is included in a PRIMARY KEY constraint.

- PRIMARY KEY
  Specifies the column that uniquely identifies a row in the table. The identified columns must be defined as NOT NULL.

  Note: If you attempt to add a primary key using ALTER TABLE and any of the columns included in the primary key contain null values, an error will be generated and the primary key will not be added. See "ALTER TABLE" on page 9 for more information.

- UNIQUE
  Specifies that values in the column must be unique. NULL values are not allowed.

- FOREIGN KEY
  Specifies that the values in the column must correspond to values in a referenced primary key or unique key column or that they are NULL.

- CHECK
  Specifies rules for values in the column.

Table constraints include:

- PRIMARY KEY
  Specifies the column or columns that uniquely identify a row in the table. NULL values are not allowed.

- UNIQUE
  Specifies that values in the columns must be unique. The identified columns must be defined as NOT NULL.

- FOREIGN KEY
Specifies that the values in the columns must correspond to values in referenced primary key or unique columns or that they are NULL.

Note: If the foreign key consists of multiple columns, and any column is NULL, the whole key is considered NULL. The insert is permitted no matter what is on the non-null columns.

- CHECK
  Specifies a wide range of rules for values in the table.

Column constraints and table constraints have the same function; the difference is in where you specify them. Table constraints allow you to specify more than one column in a PRIMARY KEY, UNIQUE, CHECK, or FOREIGN KEY constraint definition. Column-level constraints (except for CHECK constraints) refer to only one column.

**Syntax**

**Column-Level-Constraint:**

```
{ NOT NULL | [ CONSTRAINT [constraint–Name] ]
  { CHECK [searchCondition] | } [ PRIMARY KEY | UNIQUE | ReferencesSpecification ] }
```

**Table-Level Constraint:**

```
[ CONSTRAINT [constraint–Name] ]
  { CHECK [searchCondition] | }
```

**ReferencesSpecification:**

```
REFERENCES table–Name [ ( Simple–column–Name [ , Simple–column–Name ]* ) ]
[ ON DELETE { NO ACTION | RESTRICT | CASCADE | SET NULL } ]
[ ON UPDATE { NO ACTION | RESTRICT } ]
```

**searchCondition:** A searchCondition is any Boolean expression that meets the requirements specified in “Requirements for Search Condition” on page 46.

If a constraint–Name is not specified, Cloudscape generates a unique constraint name (for either column or table constraints).

**Primary Key and Unique Constraints**

A primary key defines the set of columns that uniquely identifies rows in a table.
When you create a primary key constraint, none of the columns included in the primary key can have NULL constraints; that is, they must not permit NULL values.

ALTER TABLE ADD PRIMARY KEY allows you to include existing columns in a primary key if they were first defined as NOT NULL. NULL values are not allowed. If the column(s) contain NULL values, the system will not add the primary key constraint. See "ALTER TABLE" on page 9 for more information.

A table can have at most one PRIMARY KEY constraint, but can have multiple UNIQUE constraints.

**Foreign Key Constraints**

Foreign keys provide a way to enforce the referential integrity of a database. A foreign key is a column or group of columns within a table that references a key in some other table (or sometimes, though rarely, the same table). The foreign key must always include the columns of which the types exactly match those in the referenced primary key or unique constraint.

For a table-level foreign key constraint in which you specify the columns in the table that make up the constraint, you cannot use the same column more than once.

If there is a column list in the ReferencesSpecification (a list of columns in the referenced table), it must correspond either to a unique constraint or to a primary key constraint in the referenced table. The ReferencesSpecification can omit the column list for the referenced table if that table has a declared primary key.

If there is no column list in the ReferencesSpecification and the referenced table has no primary key, a statement exception is thrown. (This means that if the referenced table has only unique keys, you must include a column list in the ReferencesSpecification.)

A foreign key constraint is satisfied if there is a matching value in the referenced unique or primary key column. If the foreign key consists of multiple columns, the foreign key value is considered NULL if any of its columns contains a NULL.

**Note:** It is possible for a foreign key consisting of multiple columns to allow one of the columns to contain a value for which there is no matching value in the referenced columns, per the SQL-92 standard. To avoid this situation, create NOT NULL constraints on all of the foreign key’s columns.

**Foreign Key Constraints and DML**

When you insert into or update a table with an enabled foreign key constraint, Cloudscape checks that the row does not violate the foreign key constraint by looking up the corresponding referenced key in the referenced table. If the constraint is not satisfied, Cloudscape rejects the insert or update with a statement exception.

When you update or delete a row in a table with a referenced key (a primary or unique constraint referenced by a foreign key), Cloudscape checks every foreign key constraint that references the key to make sure that the removal or modification of the row does not cause a constraint violation. If removal or modification of the row would cause a constraint violation, the update or delete is not permitted and Cloudscape throws a statement exception.
Cloudscape performs constraint checks at the time the statement is executed, not when the transaction commits.

**Backing Indexes**

UNIQUE, PRIMARY KEY, and FOREIGN KEY constraints generate indexes that enforce or “back” the constraint (and are sometimes called *backing indexes*). UNIQUE and PRIMARY KEY constraints generate unique indexes. FOREIGN KEY constraints generate non-unique indexes. Therefore, if a column or set of columns has a UNIQUE, PRIMARY KEY, or FOREIGN KEY constraint on it, you do not need to create an index on those columns for performance. Cloudscape has already created it for you. See “Indexes and Constraints” on page 15.

For backing indexes, you can specify only page size.

These indexes are available to the optimizer for query optimization (see “CREATE INDEX statement” on page 15) and have system-generated names.

You cannot drop backing indexes with a DROP INDEX statement; you must drop the constraint or the table.

**Check Constraints**

A check constraint can be used to specify a wide range of rules for the contents of a table. A search condition (which is a boolean expression) is specified for a check constraint. This search condition must be satisfied for all rows in the table. The search condition is applied to each row that is modified on an INSERT or UPDATE at the time of the row modification. The entire statement is aborted if any check constraint is violated.

**Requirements for Search Condition**

If a check constraint is specified as part of a column-definition, a column reference can only be made to the same column. Check constraints specified as part of a table definition can have column references identifying columns previously defined in the CREATE TABLE statement.

The search condition must always return the same value if applied to the same values. Thus, it cannot contain any of the following:

- Dynamic parameters (?)
- Date/Time Functions (CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP)
- Subqueries
- User Functions (such as USER, SESSION_USER, CURRENT_USER)

**Referential Actions**

You can specify an ON DELETE clause and/or an ON UPDATE clause, followed by the appropriate action (CASCADE, RESTRICT, SET NULL, or NO ACTION) when defining foreign keys. These clauses specify whether Cloudscape should modify corresponding foreign key values or disallow the operation, to keep foreign key relationships intact when a primary key value is updated or deleted from a table.

You specify the update and delete rule of a referential constraint when you define the referential constraint.

The update rule applies when a row of either the parent or dependent table is updated. The choices are NO ACTION and RESTRICT.
When a value in a column of the parent table’s primary key is updated and the update rule has been specified as RESTRICT, Cloudscape checks dependent tables for foreign key constraints. If any row in a dependent table violates a foreign key constraint, the transaction is rolled back.

If the update rule is NO ACTION, Cloudscape checks the dependent tables for foreign key constraints after all deletes have been executed but before triggers have been executed. If any row in a dependent table violates a foreign key constraint, the statement is rejected.

When a value in a column of the dependent table is updated, and that value is part of a foreign key, NO ACTION is the implicit update rule. NO ACTION means that if a foreign key is updated with a non-null value, the update value must match a value in the parent table’s primary key when the update statement is completed. If the update does not match a value in the parent table’s primary key, the statement is rejected.

The delete rule applies when a row of the parent table is deleted and that row has dependents in the dependent table of the referential constraint. If rows of the dependent table are deleted, the delete operation on the parent table is said to be propagated to the dependent table. If the dependent table is also a parent table, the action specified applies, in turn, to its dependents.

The choices are NO ACTION, RESTRICT, CASCADE, or SET NULL. SET NULL can be specified only if some column of the foreign key allows null values.

If the delete rule is:

NO ACTION, Cloudscape checks the dependent tables for foreign key constraints after all deletes have been executed but before triggers have been executed. If any row in a dependent table violates a foreign key constraint, the statement is rejected.

RESTRICT, Cloudscape checks dependent tables for foreign key constraints. If any row in a dependent table violates a foreign key constraint, the transaction is rolled back.

CASCADE, the delete operation is propagated to the dependent table (and that table’s dependents, if applicable).

SET NULL, each nullable column of the dependent table’s foreign key is set to null. (Again, if the dependent table also has dependent tables, nullable columns in those tables’ foreign keys are also set to null.)

Each referential constraint in which a table is a parent has its own delete rule; 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. Similarly, a row cannot be deleted if the deletion cascades to any of its descendants that are dependents in a referential constraint with the delete rule of RESTRICT or NO ACTION.

Deleting a row from the parent table involves other tables. Any table involved in a delete operation on the parent table is said to be delete-connected to the parent table. The delete can affect rows of these tables in the following ways:

• If the delete rule is RESTRICT or NO ACTION, a dependent table is involved in the operation but is not affected by the operation. (That is, Cloudscape checks the values within the table, but does not delete any values.)
• If the delete rule is SET NULL, a dependent table's rows can be updated when a row of the parent table is the object of a delete or propagated delete operation.
• If the delete rule is CASCADE, a dependent table's rows can be deleted when a parent table is the object of a delete.
• If the dependent table is also a parent table, the actions described in this list apply, in turn, to its dependents.

Examples
-- column-level primary key constraint named OUT_TRAY_PK:
CREATE TABLE SAMP.OUT_TRAY
(  
  SENT_TIMESTAMP,
  DESTINATION CHAR(8),
  SUBJECT CHAR(64) NOT NULL CONSTRAINT OUT_TRAY_PK PRIMARY KEY,
  NOTE_TEXT VARCHAR(3000)
);

-- the table-level primary key definition allows you to
-- include two columns in the primary key definition:
CREATE TABLE SAMP.SCHED
(  
  CLASS_CODE CHAR(7) NOT NULL,
  DAY SMALLINT NOT NULL,
  STARTING_TIME,
  ENDING_TIME,
  PRIMARY KEY (CLASS_CODE, DAY)
);

-- Use a column-level constraint for an arithmetic check
-- Use a table-level constraint
-- to make sure that an employee's taxes does not
-- exceed the bonus
CREATE TABLE SAMP.EMP
(  
  EMPNO CHAR(6) NOT NULL CONSTRAINT EMP_PK PRIMARY KEY,
  FIRST_NAME CHAR(12) NOT NULL,
  MIDINIT VARCHAR(12) NOT NULL,
  LAST_NAME VARCHAR(15) NOT NULL,
  SALARY DECIMAL(9,2) CONSTRAINT SAL_CK CHECK (SALARY >= 10000),
  BONUS DECIMAL(9,2),
  TAX DECIMAL(9,2),
  CONSTRAINT BONUS_CK CHECK (BONUS > TAX)
);

-- use a check constraint to allow only appropriate
-- abbreviations for the meals
CREATE TABLE FLIGHTS
(  
  FLIGHT_ID CHAR(6) NOT NULL,
  SEGMENT_NUMBER INTEGER NOT NULL,
  ORIG_AIRPORT CHAR(3),
  DEPART_TIME TIME,
  DEST_AIRPORT CHAR(3),
  ARRIVE_TIME TIME,
  MEAL CHAR(1) CONSTRAINT MEAL_CONSTRAINT CHECK (MEAL IN ('B', 'L', 'D', 'S')),
  PRIMARY KEY (FLIGHT_ID, SEGMENT_NUMBER)
);

CREATE TABLE METROPOLITAN
(  

HOTEL_ID INT NOT NULL CONSTRAINT HOTELS_PK PRIMARY KEY,
HOTEL_NAME VARCHAR(40) NOT NULL,
CITY_ID INT CONSTRAINT METRO_FK REFERENCES CITIES
);

-- create a table with a table-level primary key constraint
-- and a table-level foreign key constraint
CREATE TABLE FLTAVAIL
(
FLIGHT_ID CHAR(6) NOT NULL,
SEGMENT_NUMBER INT NOT NULL,
FLIGHT_DATE DATE NOT NULL,
ECONOMY_SEATS_TAKEN INT,
BUSINESS_SEATS_TAKEN INT,
FIRSTCLASS_SEATS_TAKEN INT,
CONSTRAINT FLTAVAIL_PK PRIMARY KEY (FLIGHT_ID, SEGMENT_NUMBER),
CONSTRAINT FLTS_FK
FOREIGN KEY (FLIGHT_ID, SEGMENT_NUMBER)
REFERENCES Flights (FLIGHT_ID, SEGMENT_NUMBER)
);

-- add a unique constraint to a column
ALTER TABLE SAMP.PROJECT
ADD CONSTRAINT P_UC UNIQUE (PROJNAME);

-- create a table whose city_id column references the
-- primary key in the Cities table
-- using a column-level foreign key constraint
CREATE TABLE CONDOS
(
CONDO_ID INT NOT NULL CONSTRAINT hotels_PK PRIMARY KEY,
CONDO_NAME VARCHAR(40) NOT NULL,
CITY_ID INT CONSTRAINT city_foreign_key
REFERENCES Cities ON DELETE CASCADE ON UPDATE RESTRICT
);

**Statement Dependency System**

INSERT and UPDATE statements depend on all constraints on the target table. DELETEs depend on unique, primary key, and foreign key constraints. These statements are invalidated if a constraint is added to or dropped from the target table.
DECLARE GLOBAL TEMPORARY TABLE statement

The DECLARE GLOBAL TEMPORARY TABLE statement defines a temporary table for the current connection. These tables do not reside in the system catalogs and are not persistent. Temporary tables exist only during the connection that declared them and cannot be referenced outside of that connection. When the connection closes, the rows of the table are deleted, and the in-memory description of the temporary table is dropped.

Temporary tables are useful when:
- the table structure is not known before using an application.
- other users do not need the same table structure.
- data in the temporary table is needed while using the application.
- the table can be declared and dropped without holding the locks on the system catalog.

Syntax:

```
DECLARE GLOBAL TEMPORARY TABLE "table–Name"
  { column–definition [ , column–definition ] * } 
[ ON COMMIT { DELETE | PRESERVE} ROWS ]
NOT LOGGED [ON ROLLBACK DELETE ROWS]
```

**table–Name:**

Names the temporary table. If a schema–Name other than SESSION is specified, an error will occur (SQLSTATE 428EK). If the schema–Name is not specified, SESSION is assigned. Multiple connections can define declared global temporary tables with the same name because each connection has its own unique table descriptor for it.

Using SESSION as the schema name of a physical table will not cause an error, but is discouraged. The SESSION schema name should be reserved for the temporary table schema.

**column–definition:**

See "column–definition" on page 20 for CREATE TABLE for more information on column–definition. DECLARE GLOBAL TEMPORARY TABLE does not allow generated-column-spec in the column–definition.

**Data-type**

Supported data-types are:
- BIGINT
- CHAR
- DATE
- DECIMAL
- DOUBLE PRECISION
- FLOAT
- INTEGER
- NUMERIC
- REAL
- SMALLINT
- TIME
• TIMESTAMP
• VARCHAR

**ON COMMIT**
Specifies the action taken on the global temporary table when a COMMIT operation is performed.

**DELETE ROWS:**
All rows of the table will be deleted if no hold-able cursor is open on the table. This is the default value for ON COMMIT. If you specify ON ROLLBACK DELETE ROWS, this will delete all the rows in the table only if the temporary table was used. ON COMMIT DELETE ROWS will delete the rows in the table even if the table was not used (if the table does not have hold-able cursors open on it).

**PRESERVE ROWS:**
The rows of the table will be preserved.

**NOT LOGGED**
Specifies the action taken on the global temporary table when a rollback operation is performed. When a ROLLBACK (or ROLLBACK TO SAVEPOINT) operation is performed, if the table was created in the unit of work (or savepoint), the table will be dropped. If the table was dropped in the unit of work (or savepoint), the table will be restored with no rows.

**ON ROLLBACK DELETE ROWS:**
This is the default value for NOT LOGGED. NOT LOGGED [ON ROLLBACK DELETE ROWS ]] specifies the action that is to be taken on the global temporary table when a ROLLBACK or (ROLLBACK TO SAVEPOINT) operation is performed. If the table data has been changed, all the rows will be deleted.

**Examples:**
```sql
set schema myapp;
create table t1(c1 int, c12 date);
declare global temporary table SESSION.t1(c1 int) not logged;
-- The SESSION qualification is redundant here because temporary -- tables can only exist in the SESSION schema.
declare global temporary table t2(c21 int) not logged;
-- The temporary table is not qualified here with SESSION because temporary -- tables can only exist in the SESSION schema.
insert into SESSION.t1 values (1);
-- SESSION qualification is mandatory here if you want to use -- the temporary table, because the current schema is "myapp."
select * from t1;
-- This select statement is referencing the "myapp.t1" physical -- table since the table was not qualified by SESSION.
```

Note that temporary tables can only be declared in the SESSION schema. You should never declare a physical schema with the SESSION name.
The following is a list of DB2 UDB DECLARE GLOBAL TEMPORARY TABLE functions that are not supported by Cloudscape:

- IDENTITY column-options
- IDENTITY attribute in copy-options
- AS (fullselect) DEFINITION ONLY
- NOT LOGGED ON ROLLBACK PRESERVE ROWS
- IN tablespace-name
- PARTITIONING KEY
- WITH REPLACE

**Restrictions on Declared Global Temporary Tables:**

Temporary tables cannot be specified in the following statements:

- ALTER TABLE
- CREATE TRIGGER
- CREATE VIEW
- LOCK
- RENAME

Temporary tables cannot be specified in referential constraints.

There is no check constraints support for columns.

The following data types cannot be used with Declared Global Temporary Tables:

- BLOB
- CLOB
- LONG VARCHAR

Temporary tables cannot be referenced in a triggered–SQL–statement.

If a statement performing an insert, update, or delete to the temporary table encounters an error, all the rows of the table are deleted.

**Restrictions Specific to Cloudscape:**

Cloudscape does not support the following on temporary tables:

- index support
- triggers and views on SESSION schema tables (including physical tables and temporary tables)
- LOCK TABLE
- constraints and primary keys
- generated-column-spec
- importing into temporary tables

Any statements referencing SESSION schema tables and views will not be cached.

**DELETE statement**

**Syntax:**
The first syntactical form, called a searched delete, removes all rows identified by the table name and WHERE clause.

The second syntactical form, called a positioned delete, deletes the current row of an open, updatable cursor. If there is no current row or if it no longer satisfies the cursor’s query, an exception is raised. For more information about updatable cursors, see “SELECT statement” on page 74.

Examples:

```
DELETE FROM SAMP.IN_TRAY;
```

```
stmt.executeUpdate("DELETE FROM SAMP.IN_TRAY WHERE CURRENT OF "+
resultSet.getCursorName())
```

A searched delete statement depends on the table being updated, all of its conglomerates (units of storage such as heaps or indexes), and any other table named in the WHERE clause. A CREATE or DROP INDEX statement for the target table of a prepared searched delete statement invalidates the prepared searched delete statement.

The positioned delete statement depends on the cursor and any tables the cursor references. You can compile a positioned delete even if the cursor has not been opened yet. However, removing the open cursor with the JDBC close method invalidates the positioned delete.

A CREATE or DROP INDEX statement for the target table of a prepared positioned delete invalidates the prepared positioned delete statement.
FOR UPDATE clause

The FOR UPDATE clause is an optional part of a SELECT statement. The FOR UPDATE clause specifies whether the ResultSet of a simple SELECT statement that meets the requirements for a cursor is updatable or not. For more information about updatability, see Requirements for Updatable Cursors.

Syntax

```
FOR
{
    READ ONLY | FETCH ONLY |
    UPDATE [ OF Simple-column-Name [ , Simple-column-Name]* ]
}
```

Simple-column-Name refers to the names visible for the table specified in the FROM clause of the underlying query.

Cursors are read-only by default. For a cursor to be updatable, you must specify FOR UPDATE.

The optimizer is able to use an index even if the column in the index is being updated. For more information about how indexes affect cursors, see Tuning IBM Cloudscape.

Examples

```
SELECT RECEIVED, SOURCE, SUBJECT, NOTE_TEXT FROM SAMP.IN_TRAY FOR UPDATE;
```
FROM clause

The FROM clause is a mandatory clause in a SelectExpression. It specifies the tables (TableExpression) from which the other clauses of the query can access columns for use in expressions.

Syntax
FROM TableExpression [, TableExpression ] *

Examples
SELECT Cities.city_id
FROM Cities
WHERE city_id < 5
-- other types of TableExpressions
SELECT TABLENAME, ISINDEX
FROM SYS.SYSTABLES T, SYS.SYSCONGLOMERATES C
WHERE T.TABLEID = C.TABLEID
ORDER BY TABLENAME, ISINDEX;
-- force the join order
SELECT *
FROM Flights, FlightAvailability
WHERE FlightAvailability.flight_id = Flights.flight_id
AND FlightAvailability.segment_number = Flights.segment_number
AND Flights.flight_id < 'AA1115'
-- a TableExpression can be a joinOperation. Therefore
-- you can have multiple join operations in a FROM clause
SELECT COUNTRIES.COUNTRY, CITIES.CITY_NAME, FLIGHTS.DEST_AIRPORT
FROM COUNTRIES LEFT OUTER JOIN CITIES
ON COUNTRIES.COUNTRY_ISO_CODE = CITIES.COUNTRY_ISO_CODE
LEFT OUTER JOIN FLIGHTS
ON Cities.AIRPORT = FLIGHTS.DEST_AIRPORT;
GROUP BY clause

A GROUP BY clause, part of a `SelectExpression` groups a result into subsets that have matching values for one or more columns. In each group, no two rows have the same value for the grouping column or columns. NULLs are considered equivalent for grouping purposes.

You typically use a GROUP BY clause in conjunction with an aggregate expression.

Syntax

```
GROUP BY [Column-Name] [, Column-Name] *
```

`Column-Name` must be a column from the current scope of the query; there can be no columns from a query block outside the current scope. For example, if a GROUP BY clause is in a subquery, it cannot refer to columns in the outer query.

`SelectItems` in the `SelectExpression` with a GROUP BY clause must contain only aggregates or grouping columns.

Examples

```
-- find the average flying_times of flights grouped by
-- airport
SELECT AVG (flying_time), orig_airport
FROM Flights
GROUP BY orig_airport

SELECT MAX(city), region
FROM Cities, Countries
WHERE Cities.country_ISO_code = Countries.country_ISO_code
GROUP BY region
-- group by an a smallint
SELECT ID, AVG(SALARY)
FROM SAMP.STAFF
GROUP BY ID
-- Get the AVGSALARY and EMPCOUNT columns, and the DEPTNO column using the AS clause
-- And group by the WORKDEPT column using the correlation name OTHERS
SELECT OTHERS.WORKDEPT AS DEPTNO,
AVG(OTHERS.SALARY) AS AVGSALARY,
COUNT(*) AS EMPCOUNT
FROM SAMP.EMPLOYEE OTHERS
GROUP BY OTHERS.WORKDEPT;
```
HAVING clause

A HAVING clause restricts the results of a GROUP BY in a SelectExpression. The HAVING clause is applied to each group of the grouped table, much as a WHERE clause is applied to a select list. If there is no GROUP BY clause, the HAVING clause is applied to the entire result as a single group. The SELECT clause cannot refer directly to any column that does not have a GROUP BY clause. It can, however, refer to constants, aggregates, and special registers.

Syntax

HAVING searchCondition

The searchCondition, which is a specialized booleanExpression, can contain only grouping columns (see “GROUP BY clause” on page 56), columns that are part of aggregate expressions, and columns that are part of a subquery. For example, the following query is illegal, because the column SALARY is not a grouping column, it does not appear within an aggregate, and it is not within a subquery:

```sql
-- SELECT COUNT(*)
-- FROM SAMP.STAFF
-- GROUP BY ID
-- HAVING SALARY > 15000;
```

Aggregates in the HAVING clause do not need to appear in the SELECT list. If the HAVING clause contains a subquery, the subquery can refer to the outer query block if and only if it refers to a grouping column.

Examples

```sql
-- Find the total number of economy seats taken on a flight,
-- grouped by airline,
-- only when the group has at least 2 records.
SELECT SUM(ECONOMY_SEATS_TAKEN), AIRLINE_FULL
FROM FLIGHTAVAILABILITY, AIRLINES
WHERE SUBSTR(FLIGHTAVAILABILITY.FLIGHT_ID, 1, 2) = AIRLINE
GROUP BY AIRLINE_FULL
HAVING COUNT(*) > 1
```
INNER JOIN

An INNER JOIN is a [JOIN operation] that allows you to specify an explicit join clause.

Syntax

```
TableExpression [ INNER ] JOIN TableExpression
{ ON booleanExpression }
```

You can specify the join clause by specifying ON with a boolean expression.

The scope of expressions in the ON clause includes the current tables and any tables in outer query blocks to the current SELECT. In the following example, the ON clause refers to the current tables:

```
SELECT *
FROM SAMP.EMPLOYEE INNER JOIN SAMP.STAFF
ON EMPLOYEE.SALARY < STAFF.SALARY;
```

The ON clause can reference tables not being joined and does not have to reference either of the tables being joined (though typically it does).

Examples

-- Join the EMP_ACT and EMPLOYEE tables
-- select all the columns from the EMP_ACT table and
-- add the employee's surname (LASTNAME) from the EMPLOYEE table
-- to each row of the result
SELECT SAMP.EMP_ACT.*, LASTNAME
FROM SAMP.EMP_ACT JOIN SAMP.EMPLOYEE
  ON EMP_ACT.EMPNO = EMPLOYEE.EMPNO;
-- Join the EMPLOYEE and DEPARTMENT tables,
-- select the employee number (EMPNO),
-- employee surname (LASTNAME),
-- department number (WORKDEPT in the EMPLOYEE table and DEPTNO in the
-- DEPARTMENT table)
-- and department name (DEPTNAME)
-- of all employees who were born (BIRTHDATE) earlier than 1930.
SELECT EMPNO, LASTNAME, WORKDEPT, DEPTNAME
FROM SAMP.EMPLOYEE JOIN SAMP.DEPARTMENT
  ON WORKDEPT = DEPTNO
  AND YEAR(BIRTHDATE) < 1930;

-- Another example of "generating" new data values,
-- using a query which selects from a VALUES clause (which is an
-- alternate form of a fullselect).
-- This query shows how a table can be derived called "X"
-- having 2 columns "R1" and "R2" and 1 row of data
SELECT *
FROM (VALUES (3, 4), (1, 5), (2, 6))
AS VALUESTABLE1(C1, C2)
JOIN (VALUES (3, 2), (1, 2),
      (0, 3)) AS VALUESTABLE2(c1, c2)
ON VALUESTABLE1.c1 = VALUESTABLE2.c1;

This results in:

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
<th>C1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

-- List every department with the employee number and
-- last name of the manager

SELECT DEPTNO, DEPTNAME, EMPNO, LASTNAME 
FROM DEPARTMENT INNER JOIN EMPLOYEE
ON MGRNO = EMPNO;

-- List every employee number and last name
-- with the employee number and last name of their manager
SELECT E.EMPNO, E.LASTNAME, M.EMPNO, M.LASTNAME 
FROM EMPLOYEE E INNER JOIN 
DEPARTMENT INNER JOIN EMPLOYEE M
  ON MGRNO = M.EMPNO
  ON E.WORKDEPT = DEPTNO;
**INSERT statement**

An INSERT statement creates a row or rows and stores them in the named table. The number of values assigned in an INSERT statement must be the same as the number of specified or implied columns.

**Syntax**

```
INSERT INTO table-Name
   [ (Simple-column-Name [ , Simple-column-Name] * ) ]
   Query
```

Query can be:

- a **SelectExpression**
- a VALUES list
- a multiple-row VALUES expression

Single-row and multiple-row lists can include the keyword DEFAULT. Specifying DEFAULT for a column inserts the column’s default value into the column. Another way to insert the default value into the column is to omit the column from the column list and only insert values into other columns in the table. For more information see "VALUES expression" on page 80.

- UNION expressions

For more information about Query, see “Query” on page 67.

**Examples**

```
INSERT INTO COUNTRIES
   VALUES ('Taiwan', 'TW', 'Asia');
```

```
-- Insert a new department into the DEPARTMENT table,
-- but do not assign a manager to the new department
INSERT INTO DEPARTMENT (DEPTNO, DEPTNAME, ADMRDEPT)
   VALUES ('E31', 'ARCHITECTURE', 'E01');
```

```
-- Insert two new departments using one statement
-- into the DEPARTMENT table as in the previous example,
-- but do not assign a manager to the new department.
INSERT INTO DEPARTMENT (DEPTNO, DEPTNAME, ADMRDEPT)
   VALUES ('E31', 'DATABASE ADMINISTRATION', 'E01'),
   ('E41', 'DATABASE ADMINISTRATION', 'E01');
```

```
-- Create a temporary table MA_EMP_ACT with the
-- same columns as the EMP_ACT Table.
-- Load MA_EMP_ACT with the rows from the EMP_ACT
-- table with a project number (PROJNO)
-- starting with the letters 'MA'.
CREATE TABLE MA_EMP_ACT
   (EMPNO CHAR(6) NOT NULL,
    PROJNO CHAR(6) NOT NULL,
    ACTNO SMALLINT NOT NULL,
    EMPTIME DEC(5,2),
    EMSTDATE DATE,
    EMENDATE DATE
   );
```

```
INSERT INTO MA_EMP_ACT
   SELECT * FROM EMP_ACT
   WHERE SUBSTR(PROJNO, 1, 2) = 'MA';
```

```
-- Insert the DEFAULT value for the LOCATION column
INSERT INTO DEPARTMENT
   VALUES ('E31', 'ARCHITECTURE', '00390', 'E01', DEFAULT);
```
**Statement Dependency System**

The INSERT statement depends on the table being inserted into, all of the conglomerates (units of storage such as heaps or indexes) for that table, and any other table named in the query. Any statement that creates or drops an index or a constraint for the target table of a prepared INSERT statement invalidates the prepared INSERT statement.
JOIN operation

The JOIN operations, which are among the possible TableExpressions in a FROM clause, perform joins between two tables. (You can also perform a join between two tables using an explicit equality test in a WHERE clause, such as “WHERE t1.col1 = t2.col2”.)

Syntax

The JOIN operations are:

• **INNER JOIN**
  Specifies a join between two tables with an explicit join clause. See “INNER JOIN” on page 58.

• **LEFT OUTER JOIN**
  Specifies a join between two tables with an explicit join clause, preserving unmatched rows from the first table. See “LEFT OUTER JOIN” on page 63.

• **RIGHT OUTER JOIN**
  Specifies a join between two tables with an explicit join clause, preserving unmatched rows from the second table. See “RIGHT OUTER JOIN” on page 69.

In all cases, you can specify additional restrictions on one or both of the tables being joined in outer join clauses or in the WHERE clause.

JOIN Expressions and Query Optimization

For information on which types of joins are optimized, see Tuning IBM Cloudscape.
LEFT OUTER JOIN

A LEFT OUTER JOIN is one of the JOIN operations that allow you to specify a join clause. It preserves the unmatched rows from the first (left) table, joining them with a NULL row in the shape of the second (right) table.

Syntax

\[
\text{TableExpression} \text{ LEFT [ OUTER ] JOIN TableExpression}
\begin{array}{l}
  \text{ON booleanExpression}
\end{array}
\]

The scope of expressions in either the ON clause includes the current tables and any tables in query blocks outer to the current SELECT. The ON clause can reference tables not being joined and does not have to reference either of the tables being joined (though typically it does).

Examples

-- match cities to countries
SELECT CITIES.COUNTRY, REGION
FROM Countries
LEFT OUTER JOIN Cities
ON CITY_ID=CITY_ID
WHERE REGION = 'Asia';
-- use the synonymous syntax, RIGHT JOIN, to achieve exactly
-- the same results as in the example above
SELECT COUNTRIES.COUNTRY, REGION
FROM Countries
LEFT JOIN Cities
ON CITY_ID=CITY_ID;

-- Join the EMPLOYEE and DEPARTMENT tables,
-- select the employee number (EMPNO),
-- employee surname (LASTNAME),
-- department number (WORKDEPT in the EMPLOYEE table
-- and DEPTNO in the DEPARTMENT table)
-- and department name (DEPTNAME)
-- of all employees who were born (BIRTHDATE) earlier than 1930
SELECT EMPNO, LASTNAME, WORKDEPT, DEPTNAME
FROM SAMP.EMPLOYEE LEFT OUTER JOIN SAMP.DEPARTMENT
ON WORKDEPT = DEPTNO
AND YEAR(BIRTHDATE) < 1930;
-- List every department with the employee number and
-- last name of the manager,
-- including departments without a manager
SELECT DEPTNO, DEPTNAME, EMPNO, LASTNAME
FROM DEPARTMENT LEFT OUTER JOIN EMPLOYEE
ON MGRNO = EMPNO;
**LOCK TABLE statement**

Allows a user to explicitly acquire a shared or exclusive table lock on the specified table. The table lock lasts until the end of the current transaction.

Explicitly locking a table is useful for:
- avoiding the overhead of multiple row locks on a table (in other words, user-initiated lock escalation)
- avoiding deadlocks

You cannot lock system tables with this statement.

**Syntax**

```
LOCK TABLE [table-Name] IN { SHARE | EXCLUSIVE } MODE
```

Once a table is locked in either mode, a transaction does not acquire any subsequent row-level locks on a table. Replace line 13 with this: For example, if a transaction locks the entire Flights table in share mode in order to read data, a particular statement might need to lock a particular row in exclusive mode in order to update the row. However, the previous table-level lock on Hotels forces the exclusive lock to be table-level as well.

If the specified lock cannot be acquired because another connection already holds a lock on the table, a statement-level exception is raised (SQLState X0X02) after the deadlock timeout period.

**Examples**

```
-- lock the entire table in share mode to avoid
-- a large number of row locks
LOCK TABLE Flights IN SHARE MODE;
SELECT *
FROM Flights
WHERE orig_airport > '000';

-- lock the entire table in exclusive mode
-- for a transaction that will update many rows,
-- but where no single statement will update enough rows
-- acquire an exclusive table lock on the table.
-- In a row-level locking system, that transaction would
-- require a large number of locks or might deadlock.
LOCK TABLE HotelAvailability IN EXCLUSIVE MODE;
UPDATE HotelAvailability
SET rooms_taken = (rooms_taken + 2)
WHERE hotel_id = 194 AND booking_date = DATE('1998-04-10');

UPDATE HotelAvailability
SET rooms_taken = (rooms_taken + 2)
WHERE hotel_id = 194 AND booking_date = DATE('1998-04-11');

UPDATE HotelAvailability
SET rooms_taken = (rooms_taken + 2)
WHERE hotel_id = 194 AND booking_date = DATE('1998-04-12');

UPDATE HotelAvailability
SET rooms_taken = (rooms_taken + 2)
WHERE hotel_id = 194 AND booking_date = DATE('1998-04-12');

-- if a transaction needs to look at a table before
```
-- updating it, acquire an exclusive lock before
-- selecting to avoid deadlocks
LOCK TABLE People IN EXCLUSIVE MODE;
SELECT MAX(person_id) + 1 FROM PEOPLE;
-- INSERT INTO PEOPLE . . .
ORDER BY clause

The ORDER BY clause is an optional element of a SELECT statement. An ORDER BY clause allows you to specify the order in which rows appear in the ResultSet.

Syntax

ORDER BY { column-Name | ColumnPosition }
[ ASC | DESC ]
[ , column-Name | ColumnPosition
[ ASC | DESC ] ] *

ColumnPosition is an integer that identifies the number of the column in the SelectItem in the underlying Query of the SELECT statement. ColumnPosition must be greater than 0 and not greater than the number of columns in the result table. In other words, if you want to order by a column, that column must be in the select list.

column-Name refers to the names visible from the SelectItems in the underlying query of the SELECT statement. An order by column does not need to be in the select list.

ASC specifies that the results should be returned in ascending order; DESC specifies that the results should be returned in descending order. If the order is not specified, ASC is the default.

An ORDER BY clause prevents a SELECT statement from being an updatable cursor. (For more information, see “Requirements for Updatable Cursors” on page 74.)

For example, if an INTEGER column contains integers, NULL is considered greater than 1 for purposes of sorting. In other words, NULL values are sorted high.

Examples

-- order by the correlation name NATION
-- order by the correlation name NATION
SELECT CITY_NAME, COUNTRY AS NATION
FROM CITIES
ORDER BY NATION;
Query

A query creates a virtual table based on existing tables or constants built into tables.

Syntax

\{ \( \text{Query} \) | \( \text{Query} \cup \text{ALL} \text{Query} \) | \text{SelectExpression} \} VALUES expression

You can arbitrarily put parentheses around queries, or use the parentheses to control the order of evaluation of UNION operations. UNION operations are evaluated from left to right when no parentheses are present.

You can combine two queries into one using the UNION [ALL] operation. UNION builds an intermediate ResultSet with all of the rows from both queries and eliminates the duplicate rows before returning the remaining rows. UNION ALL returns all rows from both queries as the result.

Examples

-- a Select expression
SELECT *
FROM FROM ORG;

-- a subquery
SELECT *
FROM (SELECT CLASS_CODE FROM CL_SCHED) AS CS;

-- a subquery
SELECT *
FROM (SELECT CLASS_CODE FROM CL_SCHED) AS CS (CLASS_CODE);

-- a UNION
-- returns all rows from columns DEPTNUMB and MANAGER
-- in table ORG
-- and (1,2) and (3,4)
-- DEPTNUMB and MANAGER are smallint columns
SELECT DEPTNUMB, MANAGER
FROM ORG
UNION ALL
VALUES (1,2), (3,4);

-- a values expression
VALUES (1,2,3)
-- List the employee numbers (EMPNO) of all employees in the EMPLOYEE table
-- whose department number (WORKDEPT) either begins with 'E' or
-- who are assigned to projects in the EMP_ACT table
-- whose project number (PROJNO) equals 'MA2100', 'MA2110', or 'MA2112'
SELECT EMPNO
FROM EMPLOYEE
WHERE WORKDEPT LIKE 'E%'
UNION
SELECT EMPNO
FROM EMP_ACT
WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112');
-- Make the same query as in the previous example
-- and "tag" the rows from the EMPLOYEE table with 'emp' and
-- the rows from the EMP_ACT table with 'emp_act'.
-- Unlike the result from the previous example,
-- this query may return the same EMPNO more than once,
-- identifying which table it came from by the associated "tag"
SELECT EMPNO, 'emp'
      FROM EMPLOYEE
      WHERE WORKDEPT LIKE 'E%'
UNION
SELECT EMPNO, 'emp_act' FROM EMP_ACT
      WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112');
-- Make the same query as in the previous example,
-- only use UNION ALL so that no duplicate rows are eliminated
SELECT EMPNO
      FROM EMPLOYEE
      WHERE WORKDEPT LIKE 'E%'
UNION
SELECT EMPNO
      FROM EMP_ACT
      WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112');
-- Make the same query as in the previous example,
-- only include an additional two employees currently not in any table and
-- tag these rows as "new"
SELECT EMPNO, 'emp'
      FROM EMPLOYEE
      WHERE WORKDEPT LIKE 'E%'
UNION
SELECT EMPNO, 'emp_act'
      FROM EMP_ACT
      WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')
UNION
VALUES ('NEWAAA', 'new'), ('NEWBBB', 'new');
RIGHT OUTER JOIN

A RIGHT OUTER JOIN is one of the JOIN operations that allow you to specify a JOIN clause. It preserves the unmatched rows from the second (right) table, joining them with a NULL in the shape of the first (left) table. A LEFT OUTER JOIN B is equivalent to B RIGHT OUTER JOIN A, with the columns in a different order.

Syntax

```
TableExpression RIGHT [ OUTER ] JOIN TableExpression
{
  ON booleanExpression
}
```

The scope of expressions in the ON clause includes the current tables and any tables in query blocks outer to the current SELECT. The ON clause can reference tables not being joined and does not have to reference either of the tables being joined (though typically it does).

Examples

-- get all countries and corresponding cities, including
countries without any cities
SELECT CITY_NAME, CITIES.COUNTRY
FROM CITIES RIGHT OUTER JOIN COUNTRIES
ON CITIES.COUNTRY_ISO_CODE = COUNTRIES.COUNTRY_ISO_CODE;

-- get all countries in Africa and corresponding cities, including
countries without any cities
SELECT CITY_NAME, CITIES.COUNTRY
FROM CITIES RIGHT OUTER JOIN COUNTRIES
ON CITIES.COUNTRY_ISO_CODE = COUNTRIES.COUNTRY_ISO_CODE;
WHERE Countries.region = 'frica';

-- use the synonymous syntax, RIGHT JOIN, to achieve exactly
-- the same results as in the example above
SELECT CITY_NAME, CITIES.COUNTRY
FROM CITIES RIGHT JOIN COUNTRIES
ON CITIES.COUNTRY_ISO_CODE = COUNTRIES.COUNTRY_ISO_CODE
WHERE Countries.region = 'Africa';

-- a TableExpression can be a joinOperation. Therefore
-- you can have multiple join operations in a FROM clause
-- List every employee number and last name
-- with the employee number and last name of their manager
SELECT E.EMPNO, E.LASTNAME, M.EMPNO, M.LASTNAME
FROM EMPLOYEE E RIGHT OUTER JOIN
DEPARTMENT RIGHT OUTER JOIN EMPLOYEE M
  ON MGRNO = M.EMPNO
  ON E.WORKDEPT = DEPTNO;
ScalarSubquery

You can place a ScalarSubquery anywhere an Expression is permitted. A ScalarSubquery turns a SelectExpression result into a scalar value because it returns only a single row and column value.

The query must evaluate to a single row with a single column.

Sometimes also called an expression subquery.

Syntax

```
(Query)
```

Examples

-- avg always returns a single value, so the subquery is
-- a ScalarSubquery
```
SELECT NAME
FROM STAFF
(SELECT AVG(BONUS + 800)
 FROM EMPLOYEE
 WHERE COMM < 5000);
```

-- Introduce a way of "generating" new data values,
-- using a query which selects from a VALUES clause (which is an alternate form of a fullselect).
-- This query shows how a table can be derived called "X" having
2 columns "R1" and "R2" and 1 row of data.
```
SELECT R1,R2
FROM (VALUES('GROUP 1','GROUP 2')) AS X(R1,R2);
```
SelectExpression

A SelectExpression is the basic SELECT-FROM-WHERE construct used to build a table value based on filtering and projecting values from other tables.

Syntax

```
SELECT [ DISTINCT | ALL ] SelectItem [ , SelectItem ]*
FROM clause
WHERE clause
GROUP BY clause
HAVING clause
```

SelectItem::

```
{ *
  table-Name | correlation-Name] , * |
Expression [AS Simple-column-Name]
}
```

The SELECT clause contains a list of expressions and an optional quantifier that is applied to the results of the FROM clause and the WHERE clause. If DISTINCT is specified, only one copy of any row value is included in the result. Nulls are considered duplicates of one another for the purposes of DISTINCT. If no quantifier, or ALL, is specified, no rows are removed from the result in applying the SELECT clause (ALL is the default).

A SelectItem projects one or more result column values for a table result being constructed in a SelectExpression.

The result of the FROM clause is the cross product of the FROM items. The WHERE clause can further qualify this result.

The WHERE clause causes rows to be filtered from the result based on a boolean expression. Only rows for which the expression evaluates to TRUE are returned in the result.

The GROUP BY clause groups rows in the result into subsets that have matching values for one or more columns. GROUP BY clauses are typically used with aggregates.

If there is a GROUP BY clause, the SELECT clause must contain only aggregates or grouping columns. If you want to include a non-grouped column in the SELECT clause, include the column in an aggregate expression. For example:

```
-- List head count of each department,
-- the department number (WORKDEPT), and the average departmental salary (SALARY)
-- for all departments in the EMPLOYEE table.
-- Arrange the result table in ascending order by average departmental salary.
SELECT WORKDEPT, AVG(SALARY)
  FROM EMPLOYEE
  GROUP BY WORKDEPT
  ORDER BY 3;
```

If there is no GROUP BY clause, but a SelectItem contains an aggregate not in a subquery, the query is implicitly grouped. The entire table is the single group.

The HAVING clause restricts a grouped table, specifying a search condition (much like a WHERE clause) that can refer only to grouping columns or aggregates from the current scope. The HAVING clause is applied to each group of the grouped...
If the HAVING clause evaluates to TRUE, the row is retained for further processing. If the HAVING clause evaluates to FALSE or NULL, the row is discarded. If there is a HAVING clause but no GROUP BY, the table is implicitly grouped into one group for the entire table.

Cloudscape processes a SelectExpression in the following order:

- FROM clause
- WHERE clause
- GROUP BY (or implicit GROUP BY)
- HAVING clause
- SELECT clause

The result of a SelectExpression is always a table.

When a query does not have a FROM clause (when you are constructing a value, not getting data out of a table), you use a VALUES statement, not a SelectExpression. For example:

VALUES CURRENT_TIMESTAMP

See “VALUES expression” on page 80.

The * Wildcard

* expands to all columns in the tables in the associated FROM clause.

`table-Name` * and `correlation-Name` expand to all columns in the identified table. That table must be listed in the associated FROM clause.

Naming Columns

You can name a SelectItem column using the AS clause. When the SelectExpression appears in a UNION, the names from the first SelectExpression are taken as the names for the columns in the result of the UNION. If a column of a SelectItem is not a simple ColumnReference expression or named with an AS clause, it is given a generated unique name.

These column names are useful in several cases:

- They are made available on the JDBC ResultSetMetaData.
- They are used as the names of the columns in the resulting table when the SelectExpression is used as a table subquery in a FROM clause.
- They are used in the ORDER BY clause as the column names available for sorting.

Examples

-- this example shows SELECT-FROM-WHERE
-- with an ORDER BY clause
-- and correlation-Names for the tables
SELECT CONSTRAINTNAME, COLUMNNAME
FROM SYS.SYSTABLES t, SYS.SYSCOLUMNS col,
SYS.SYSCONSTRAINTS cons, SYS.SYSCHECKS checks
WHERE t.TABLENAME = 'FLIGHTS' AND t.TABLEID = col.
REFERENCEID AND t.TABLEID = cons.TABLEID
AND cons.CONSTRAINTID = checks.CONSTRAINTID
ORDER BY CONSTRAINTNAME;
-- This example shows the use of the DISTINCT clause
SELECT DISTINCT ACTNO
FROM EMP_ACT;
-- This example shows how to rename an expression

-- Using the EMPLOYEE table, list the department number (WORKDEPT) and
-- maximum departmental salary (SALARY) renamed as BOSS
-- for all departments whose maximum salary is less than the
-- average salary in all other departments.
SELECT WORKDEPT AS DPT, MAX(SALARY) AS BOSS
FROM EMPLOYEE EMP_COR
GROUP BY WORKDEPT
HAVING MAX(SALARY) < (SELECT AVG(SALARY)
FROM EMPLOYEE
WHERE NOT WORKDEPT = EMP_COR.WORKDEPT)
ORDER BY BOSS;
SELECT statement

A SELECT statement consists of a query with an optional ORDER BY clause and an optional FOR UPDATE clause. The SELECT statement is so named because the typical first word of the query construct is SELECT. (Query includes the VALUES expression and UNION expressions as well as SELECT expressions).

The ORDER BY clause guarantees the ordering of the ResultSet. The FOR UPDATE clause makes the result an updatable cursor. The SELECT statement supports the FOR FETCH ONLY clause. The FOR FETCH ONLY clause is synonymous with the FOR READ ONLY clause.

Syntax

```sql
[query]
[ORDER BY clause]
[FOR UPDATE clause]
WITH {RR|RS|CS|UR}
```

You can set the isolation level in a SELECT statement using the WITH {RR|RS|CS|UR} syntax.

Examples

-- lists the names of the expression SAL+BONUS+COMM as TOTAL_PAY and
-- orders by the new name TOTAL_PAY
```sql
SELECT FIRSTNAME, SALARY+BONUS+COMM AS TOTAL_PAY
FROM EMPLOYEE
ORDER BY TOTAL_PAY;
```

-- creating an updatable cursor with a FOR UPDATE clause
-- to update the start date (PRSTDATE) and the end date (PRENDATE)
-- columns in the PROJECT table
```sql
SELECT PROJNO, PRSTDATE, PRENDATE
FROM PROJECT
FOR UPDATE OF PRSTDATE, PRENDATE;
```

-- set the isolation level to RR for this statement only
```sql
SELECT *
FROM Flights
WHERE flight_id BETWEEN 'AA1111' AND 'AA1112'
WITH RR;
```

A SELECT statement returns a ResultSet. A cursor is a pointer to a specific row in ResultSet. In Java applications, all ResultSets are cursors. A cursor is updatable; that is, you can update or delete rows as you step through the ResultSet if the SELECT statement that generated it and its underlying query meet cursor updatability requirements, as detailed below. You use a FOR UPDATE clause when you want to generate an updatable cursor.

Note: The ORDER BY clause allows you to order the results of the SELECT.

Without the ORDER BY clause, the results are returned in random order.

If a SELECT statement meets the requirements listed below, cursors are updatable only if you specify FOR UPDATE in the FOR clause (see "FOR UPDATE clause" on page 54).

Requirements for Updatable Cursors

Only simple, single-table SELECT cursors can be made updatable. To generate updatable cursors:

- The SELECT statement must not include an ORDER BY clause.
• The underlying Query must be a `SelectExpression`.
• The `SelectExpression` in the underlying Query must not include:
  - DISTINCT
  - Aggregates
  - GROUP BY clause
  - HAVING clause
• The FROM clause in the underlying Query must not have:
  - more than one table in its FROM clause
  - anything other than one table name
  - `SelectExpressions`
  - subqueries

There is no SQL language statement to assign a name to a cursor. Instead, you use the JDBC API to assign names to cursors or retrieve system-generated names. For more information, see “Naming or Accessing the Name of a Cursor” in Chapter 5 of the IBM Cloudscape Developer’s Guide.

Cursors are read-only by default. For a cursor to be updatable, you must specify FOR UPDATE in the FOR clause (see “FOR UPDATE clause” on page 54).

**Statement Dependency System**

The SELECT depends on all the tables and views named in the query and the conglomerates (units of storage such as heaps and indexes) chosen for access paths on those tables. CREATE INDEX does not invalidate a prepared SELECT statement. A DROP INDEX statement invalidates a prepared SELECT statement if the index is an access path in the statement. If the SELECT includes views, it also depends on the dictionary objects on which the view itself depends (see “CREATE VIEW statement” on page 27).

Any prepared UPDATE WHERE CURRENT or DELETE WHERE CURRENT statement against a cursor of a SELECT depends on the SELECT. Removing a SELECT through a `java.sql.Statement.close` request invalidates the UPDATE WHERE CURRENT or DELETE WHERE CURRENT.

The SELECT depends on all aliases used in the query. Dropping an alias invalidates a prepared SELECT statement if the statement uses the alias.
TableExpression

A TableExpression specifies a table or view in a FROM clause. It is the source from which a SelectExpression selects a result.

A correlation name can be applied to a table in a TableExpression so that its columns can be qualified with that name. If you do not supply a correlation name, the table name qualifies the column name. When you give a table a correlation name, you cannot use the table name to qualify columns. You must use the correlation name when qualifying column names.

No two items in the FROM clause can have the same correlation name, and no correlation name can be the same as an unqualified table name specified in that FROM clause.

In addition, you can give the columns of the table new names in the AS clause. Some situations in which this is useful:

- When a VALUES expression is used as a TableSubquery since there is no other way to name the columns of a VALUES expression
- When column names would otherwise be the same as those of columns in other tables; renaming them means you don’t have to qualify them.

The Query in a TableSubquery appearing in a FromItem can contain multiple columns and return multiple rows. See TableSubquery.

For information about the optimizer overrides you can specify, see Tuning IBM Cloudscape.

Syntax

{ TableOrViewExpression | JOIN operation }

TableOrViewExpression:

{ Table-Name | view-Name }

Examples

-- SELECT from a Join expression
SELECT E.EMPNO, E.LASTNAME, M.EMPNO, M.LASTNAME
FROM EMPLOYEE E LEFT OUTER JOIN
   DEPARTMENT INNER JOIN EMPLOYEE M
   ON MGRNO = M.EMPNO
   ON E.WORKDEPT = DEPTNO
TableSubquery

A TableSubquery is a subquery that returns multiple rows.

Unlike a ScalarSubquery, a TableSubquery is allowed only:
- as a TableExpression in a FROM clause;
- with EXISTS, IN, or quantified comparisons (see “Quantified comparison” on page 179).

When used as a TableExpression in a FROM clause, it can return multiple columns. When used with EXISTS, it returns multiple columns only if you use * to return the multiple columns.

When used with IN or Quantified comparison, it must return a single column.

Syntax

Examples

-- a subquery used as a TableExpression in a FROM clause
SELECT VirtualFlightTable.flight_ID
FROM
  (SELECT flight_ID, orig_airport, dest_airport
   FROM Flights
   WHERE (orig_airport = 'SFO' OR dest_airport = 'SCL')
  ) AS VirtualFlightTable

-- a subquery (values expression) used as a TableExpression
-- in a FROM clause
SELECT mycol1
FROM
  (VALUES (1, 2), (3, 4))
AS mytable (mycol1, mycol2)

-- a subquery used with EXISTS
SELECT *
FROM Flights
WHERE EXISTS
  (SELECT * FROM Flights WHERE dest_airport = 'SFO'
   AND orig_airport = 'GRU')

-- a subquery used with IN
SELECT flight_id, segment_number
FROM Flights
WHERE flight_id IN
  (SELECT flight_ID
   FROM Flights WHERE orig_airport = 'SFO'
   OR dest_airport = 'SCL')

-- a subquery used with a quantified comparison
SELECT NAME, COMM
FROM STAFF
WHERE COMM >
  (SELECT AVG(BONUS + 800)
   FROM EMPLOYEE
   WHERE COMM < 5000);
UPDATE statement

An UPDATE statement sets the value in a column.

You can update the current row of an open, updatable cursor. If there is no current row, or if the current row no longer satisfies the cursor’s query, an exception is raised.

Syntax

```
{ 
  UPDATE table-Name 
  SET column-Name = Value 
  [ , column-Name = Value ]* 
  WHERE clause 

  UPDATE table-Name 
  SET column-Name = Value 
  [ , column-Name = Value ]* 
  WHERE CURRENT OF clause 
}
```

Value:

Expression | DEFAULT

The first syntactical form is called a searched update. The second syntactical form is called a positioned update.

For searched updates, you update all rows of the table for which the WHERE clause evaluates to TRUE.

For positioned updates, you can update only columns that were included in the FOR UPDATE clause of the SELECT statement that created the cursor. If the SELECT statement did not include a FOR UPDATE clause, the cursor is read-only and cannot be used to update.

Specifying DEFAULT for the update value sets the value of the column to the default defined for that table.

Examples

-- All the employees except the manager of department (WORKDEPT) 'E21' have been temporarily reassigned.
-- Indicate this by changing their job (JOB) to NULL and their pay
-- (SALARY, BONUS, COMM) values to zero in the EMPLOYEE table.
UPDATE EMPLOYEE
  SET JOB=NULL, SALARY=0, BONUS=0, COMM=0
  WHERE WORKDEPT = 'E21' AND JOB <> 'MANAGER'

-- PROMOTE the job (JOB) of certain employees to MANAGER
UPDATE EMPLOYEE
  SET JOB = 'MANAGER'
  WHERE CURRENT OF CURS1;
-- Increase the project staffing (PRSTAFF) by 1.5 for all projects
stmt.executeUpdate("UPDATE PROJECT SET PRSTAFF = "
"PRSTAFF + 1.5" +
"WHERE CURRENT OF" + ResultSet.getCursorName())

-- Change the job (JOB) of employee number (EMPNO) '000290' in the EMPLOYEE table
-- to its DEFAULT value which is NULL
UPDATE EMPLOYEE
  SET JOB = DEFAULT
  WHERE EMPNO = '000290';
**Statement Dependency System**

A searched update statement depends on the table being updated, all of its conglomerates (units of storage such as heaps or indexes), all of its constraints, and any other table named in the WHERE clause or SET expressions. A CREATE or DROP INDEX statement or an ALTER TABLE statement for the target table of a prepared searched update statement invalidates the prepared searched update statement.

The positioned update statement depends on the cursor and any tables the cursor references. You can compile a positioned update even if the cursor has not been opened yet. However, removing the open cursor with the JDBC `close` method invalidates the positioned update.

A CREATE or DROP INDEX statement or an ALTER TABLE statement for the target table of a prepared positioned update invalidates the prepared positioned update statement.

Dropping an alias invalidates a prepared update statement if the latter statement uses the alias.

Dropping or adding triggers on the target table of the update invalidates the update statement.
VALUES expression

The VALUES expression allows construction of a row or a table from other values. You use a VALUES statement when you do not have a FROM clause. This construct can be used in all the places where a query can, and thus can be used as a statement that returns a ResultSet, within expressions and statements wherever subqueries are permitted, and as the source of values for an INSERT statement.

Syntax

```
{ 
    VALUES (Value, Value)* 
    [ , (Value, Value)* ]* | 
    VALUES Value, Value)* | 
}
```

Value:

Expression | DEFAULT

The first form constructs multi-column rows. The second form constructs single-column rows, each expression being the value of the column of the row.

The DEFAULT keyword is allowed only if the VALUES expression is in an INSERT statement. Specifying DEFAULT for a column inserts the column’s default value into the column. Another way to insert the default value into the column is to omit the column from the column list and only insert values into other columns in the table.

Examples

-- 3 rows of 1 column
VALUES (1), (2), (3);
-- 3 rows of 1 column
VALUES 1, 2, 3;
-- 1 row of 3 columns
VALUES (1, 2, 3);
-- 3 rows of 2 columns
VALUES (1, 21), (2, 22), (3, 23);
-- constructing a derived table
VALUES ('orange', 'orange'), ('apple', 'red'),
       ('banana', 'yellow')

-- Insert two new departments using one statement into the DEPARTMENT table,
-- but do not assign a manager to the new department.
INSERT INTO DEPARTMENT (DEPTNO, DEPTNAME, ADMRDEPT)
VALUES ('B11', 'URCHASING', 'B01'),
       ('E41', 'DATABASE ADMINISTRATION', 'E01')
-- Insert a row with a DEFAULT value for the MAJPROJ column
INSERT INTO PROJECT (PROJNO, PROJNAME, DEPTNO, RESPEMP, PRSTDATE, MAJPROJ)
VALUES ('PL2101', 'ENSURE COMPAT PLAN', 'B01', '000020', CURRENT_DATE, DEFAULT);

-- using a built-in function
VALUES CURRENT_DATE
-- getting the value of an arbitrary expression
VALUES (3*29, 26.0E0/3)
-- getting a value returned by a built-in function
VALUES char(1)
WHERE clause

A WHERE clause is an optional part of a SelectExpression DELETE statement or UPDATE statement. The WHERE clause lets you select rows based on a boolean expression. Only rows for which the expression evaluates to TRUE are returned in the result, or, in the case of a DELETE statement, deleted, or, in the case of an UPDATE statement, updated.

Syntax

WHERE Boolean expression

Boolean expressions are allowed in the WHERE clause. Most of the general expressions listed in Table 6 on page 174 can result in a boolean value.

In addition, there are the more common boolean expressions. Specific boolean operators listed in Table 10, take one or more operands; the expressions return a boolean value.

Examples

-- find the flights where no business-class seats have
-- been booked
SELECT *
FROM FlightAvailability
WHERE business_seats_taken IS NULL
   OR business_seats_taken = 0
-- Join the EMP.ACT and EMPLOYEE tables
-- select all the columns from the EMP.ACT table and
-- add the employee's surname (LASTNAME) from the EMPLOYEE table
-- to each row of the result.
SELECT SAMP.EMP.ACT.*, LASTNAME
FROM SAMP.EMP.ACT, SAMP.EMPLOYEE
   WHERE EMP.ACT.EMPNO = EMPLOYEE.EMPNO;
-- Determine the employee number and salary of sales representatives
-- along with the average salary and head count of their departments.
-- This query must first create a new-column-name specified in the AS clause
-- which is outside the fullselect (DINFO)
-- in order to get the AVGSALARY and EMPCOUNT columns,
-- as well as the DEPTNO column that is used in the WHERE clause
SELECT THIS_EMP.EMPNO, THIS_EMP.SALARY, DINFO.AVGSALARY, DINFO.EMPCOUNT
   FROM EMPLOYEE THIS_EMP,
   (SELECT OTHERS.WORKDEPT AS DEPTNO,
      AVG(OTHERS.SALARY) AS AVGSALARY,
      COUNT(*) AS EMPCOUNT
   FROM EMPLOYEE OTHERS
   GROUP BY OTHERS.WORKDEPT
   )AS DINFO
   WHERE THIS_EMP.JOB = 'SALESREP'
      AND THIS_EMP.WORKDEPT = DINFO.DEPTNO;
WHERE CURRENT OF clause

The WHERE CURRENT OF clause is a clause in some UPDATE and DELETE statements. It allows you to perform positioned updates and deletes on updatable cursors. For more information about updatable cursors, see “SELECT statement” on page 74.

Syntax

WHERE CURRENT OF \texttt{Cursor\textasciitilde Name}

Examples

Statement \texttt{s = conn.createStatement();}
\texttt{s.setCursorName("AirlinesResults");}
\texttt{ResultSet rs = conn.executeQuery(}
   \texttt{"SELECT Airline, basic\_rate +
      "FROM Airlines FOR UPDATE OF basic\_rate");}
Statement \texttt{s2 = conn.createStatement();}
\texttt{s2.executeUpdate("UPDATE Airlines SET basic\_rate = basic\_rate +
      "+ .25 WHERE CURRENT OF AirlinesResults");}
Built-In Functions

A built-in function is an expression in which an SQL keyword or special operator executes some operation. Built-in functions use keywords or special built-in operators. Built-ins are SQL92Identifiers and are case-insensitive.

Standard Built-In Functions

- “ABS or ABSVAL” on page 86
- “BIGINT” on page 88
- “CAST” on page 89
- “LENGTH” on page 93
- “CHAR” on page 91
- “Concatenation” on page 94
- “NULLIF and CASE expressions” on page 95
- “CURRENT_DATE” on page 99
- “CURRENT ISOLATION” on page 100
- “CURRENT_TIME” on page 103
- “CURRENT_TIMESTAMP” on page 105
- “CURRENT_USER” on page 106
- “DATE” on page 107
- “DAY” on page 108
- “DOUBLE” on page 109
- “HOUR” on page 110
- “IDENTITY_VAL_LOCAL” on page 111
- “INTEGER” on page 113
- “LOCATE” on page 114
- “LCASE or LOWER” on page 115
- “LTRIM” on page 116
- “MINUTE” on page 119
- “MOD” on page 120
- “MONTH” on page 121
- “RTRIM” on page 122
- “SECOND” on page 123
- “SESSION_USER” on page 124
- “SMALLINT” on page 125
- “SQRT” on page 126
- “SUBSTR” on page 127
- “TIME” on page 129
- “TIMESTAMP” on page 130
- “UCASE or UPPER” on page 131
- “USER” on page 132
- “VARCHAR” on page 133
- “YEAR” on page 134
Aggregates (Set Functions)

This section describes aggregates (also described as set functions in ANSI SQL-92 and as column functions in some database literature). They provide a means of evaluating an expression over a set of rows. Whereas the other built-in functions operate on a single expression, aggregates operate on a set of values and reduce them to a single scalar value. Built-in aggregates can calculate the minimum, maximum, sum, count, and average of an expression over a set of values as well as count rows. You can also create your own aggregates to perform other set functions such as calculating the standard deviation.

The built-in aggregates can operate on the data types shown in Table 2.

Table 2. Permitted Data Types for Built-in Aggregates

<table>
<thead>
<tr>
<th></th>
<th>All Types</th>
<th>Numeric Built-in Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MIN</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>MAX</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>AVG</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SUM</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Aggregates are permitted only in the following:

- A SelectItem in a SelectExpression
- A HAVING clause
- An ORDER BY clause (using an alias name) if the aggregate appears in the result of the relevant query block. That is, an alias for an aggregate is permitted in an ORDER BY clause if and only if the aggregate appears in a SelectItem in a SelectExpression.

All expressions in SelectItems in the SelectExpression must be either aggregates or grouped columns (see “GROUP BY clause” on page 56). (The same is true if there is a HAVING clause without a GROUP BY clause.) This is because the ResultSet of a SelectExpression must be either a scalar (single value) or a vector (multiple values), but not a mixture of both. (Aggregates evaluate to a scalar value, and the reference to a column can evaluate to a vector.) For example, the following query mixes scalar and vector values and thus is not valid:

```sql
-- not valid
SELECT MIN(flying_time), flight_id
FROM Flights
```

Aggregates are not allowed on outer references (correlations). This means that if a subquery contains an aggregate, that aggregate cannot evaluate an expression that includes a reference to a column in the outer query block. For example, the following query is not valid because SUM operates on a column from the outer query:

```sql
SELECT c1
FROM t1
GROUP BY c1
HAVING c2 >
    (SELECT t2.x
     FROM t2
     WHERE t2.y = SUM(t1.c3))
```
A cursor declared on a ResultSet that includes an aggregate in the outer query block is not updatable.

This section includes the following aggregates:

- “AVG” on page 87
- “COUNT” on page 96
- “MAX” on page 117
- “MIN” on page 118
- “SUM” on page 128
ABS or ABSVAL

ABS or ABSVAL returns the absolute value of a numeric expression. The return type is the type of parameter. All built-in numeric types are supported (DECIMAL, DOUBLE PRECISION, FLOAT, INTEGER, BIGINT, NUMERIC, REAL, and SMALLINT).

Syntax
ABS(NumericExpression)

Examples
-- returns 3
VALUES ABS(-3)
AVG

AVG is an aggregate function that evaluates the average of an expression over a set of rows (see “Aggregates (Set Functions)” on page 84). AVG is allowed only on expressions that evaluate to numeric data types.

Syntax

AVG ( [ DISTINCT | ALL ] Expression )

The DISTINCT qualifier eliminates duplicates. The ALL qualifier retains duplicates. ALL is the default value if neither ALL nor DISTINCT is specified. For example, if a column contains the values 1.0, 1.0, 1.0, 1.0, and 2.0, AVG(col) returns a smaller value than AVG(DISTINCT col).

Only one DISTINCT aggregate expression per SelectExpression is allowed. For example, the following query is not valid:

SELECT AVG (DISTINCT flying_time), SUM (DISTINCT miles) FROM Flights

The expression can contain multiple column references or expressions, but it cannot contain another aggregate or subquery. It must evaluate to an SQL-92 numeric data type. You can therefore call methods that evaluate to SQL-92 data types. If an expression evaluates to NULL, the aggregate skips that value.

The resulting data type is the same as the expression on which it operates (it will never overflow). The following query, for example, returns the INTEGER 1, which might not be what you would expect:

SELECT AVG(c1) FROM (VALUES (1), (1), (1), (1), (2)) AS myTable (c1)

CAST the expression to another data type if you want more precision:

SELECT AVG(CAST (c1 AS DOUBLE PRECISION)) FROM (VALUES (1), (1), (1), (1), (2)) AS myTable (c1)
**BIGINT**

The BIGINT function returns a 64-bit integer representation of a number or character string in the form of an integer constant.

**Syntax**

```
BIGINT (CharacterExpression | NumericExpression )
```

**CharacterExpression**

An expression that returns a character string value of length not greater than the maximum length of a character constant. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an SQL integer constant. The character string cannot be a long string. If the argument is a CharacterExpression, the result is the same number that would occur if the corresponding integer constant were assigned to a big integer column or variable.

**NumericExpression**

An expression that returns a value of any built-in numeric data type. If the argument is a NumericExpression, 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 integers, an error occurs. The decimal part of the argument is truncated if present.

The result of the function is a big integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

**Examples**

Using the EMPLOYEE table, select the EMPNO column in big integer form for further processing in the application:

```
SELECT BIGINT (EMPNO) FROM EMPLOYEE
```
CAST

CAST converts a value from one data type to another and provides a data type to a dynamic parameter (?) or a NULL value.

CAST expressions are permitted anywhere expressions are permitted.

Syntax

\[
\text{CAST \ ( \ [ \text{Expression} \ | \ \text{NULL} \ | \ ? \ ] \ AS \ Datatype)}
\]

The data type to which you are casting an expression is the target type. The data type of the expression from which you are casting is the source type.

CAST Conversions Among SQL-92 Data Types

If a conversion is valid, CASTs are allowed. Size incompatibilities between the source and target types might cause runtime errors.

Notes

In this discussion, the Cloudscape SQL-92 data types are categorized as follows:

- numeric
  - exact numeric (SMALLINT, INTEGER, BIGINT, DECIMAL, NUMERIC)
  - approximate numeric (FLOAT, REAL, DOUBLE PRECISION)
- string
  - character string (CLOB, CHAR, VARCHAR, LONG VARCHAR)
  - bit string (BLOB, CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, LONG VARCHAR FOR BIT DATA)
- date/time
  - DATE
  - TIME
  - TIMESTAMP

Conversions from Numeric Types

A numeric type can be converted to any other numeric type. If the target type cannot represent the non-fractional component without truncation, an exception is raised. If the target numeric cannot represent the fractional component (scale) of the source numeric, then the source is silently truncated to fit into the target. For example, casting 763.1234 as INTEGER yields 763.

Conversions from and to Bit Strings

Bit strings can be converted to other bit strings, but not character strings. Strings that are converted to bit strings are padded with trailing zeros to fit the size of the target bit string. The BLOB type is more limited and requires explicit casting. In most cases the BLOB type cannot be casted to and from other types.

Conversions of Date/Time Values

A date/time value can always be converted to and from a TIMESTAMP. If a DATE is converted to a TIMESTAMP, the TIME component of the resulting TIMESTAMP is always 00:00:00. If a TIME data value is converted to a TIMESTAMP, the DATE component is set to the value of CURRENT_DATE at the time the CAST is executed. If a TIMESTAMP is converted to a DATE, the TIME component is silently truncated. If a TIMESTAMP is converted to a TIME, the DATE component is silently truncated.
Example

```sql
SELECT CAST (miles AS INT)
FROM Flights
-- convert timestamps to text
INSERT INTO mytable (text_column)
VALUES (CAST (CURRENT_TIMESTAMP AS VARCHAR(100)))
-- you must cast NULL as a data type to use it
SELECT airline
FROM Airlines
UNION ALL
VALUES (CAST (NULL AS CHAR(2)))
-- cast a double as a decimal
SELECT CAST (FLYING_TIME AS DECIMAL(5,2))
FROM FLIGHTS
-- cast a SMALLINT to a BIGINT
VALUES CAST (CAST (12 as SMALLINT) as BIGINT)
```
CHAR

The CHAR function returns a fixed-length character string representation of:

- a character string, if the first argument is any type of character string.
- a datetime value, if the first argument is a date, time, or timestamp.
- a decimal number, if the first argument is a decimal number.
- a double-precision floating-point number, if the first argument is a DOUBLE or REAL.
- an integer number, if the first argument is a SMALLINT, INTEGER, or BIGINT.

The first argument must be of a built-in data type. The result of the function is a fixed-length character string. If the first argument can be null, the result can be null. If the first argument is null, the result is the null value.

Syntax

Character to character::

CHAR (CharacterExpression [, integer] )

CharacterExpression

An expression that returns a value that is CHAR, VARCHAR, LONG VARCHAR, or CLOB data type.

integer

The length attribute for the resulting fixed length character string. The value must be between 0 and 254.

If the length of the character-expression is less than the length attribute of the result, the result is padded with blanks up to the length of the result. If the length of the character-expression is greater than the length attribute of the result, truncation is performed. A warning is returned unless the truncated characters were all blanks and the character-expression was not a long string (LONG VARCHAR or CLOB).

Integer to character::

CHAR (IntegerExpression )

IntegerExpression

An expression that returns a value that is an integer data type (either SMALLINT, INTEGER or BIGINT).

The result is the 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 with a preceding minus sign if the argument is negative. It is left justified.

- If the first argument is a small integer: The length of the result is 6. If the number of characters in the result is less than 6, then the result is padded on the right with blanks to length 6.
- If the first argument is a large integer: The length of the result is 11. If the number of characters in the result is less than 11, then the result is padded on the right with blanks to length 11.
- If the first argument is a big integer: The length of the result is 20. If the number of characters in the result is less than 20, then the result is padded on the right with blanks to length 20.

Datetime to character::
CHAR ( DatetimeExpression )

DatetimeExpression
An expression that is one of the following three data types:
- date: The result is the character representation of the date. The length of
  the result is 10.
- time: The result is the character representation of the time. The length of
  the result is 8.
- timestamp: The result is the character string representation of the
  timestamp. The length of the result is 26.

Decimal to character::
CHAR ( DecimalExpression )

DecimalExpression
An expression that returns a value that is a decimal data type. If a different
precision and scale is desired, the DECIMAL scalar function can be used
first to make the change.

Floating-point to character::
CHAR ( FloatingPointExpression )

FloatingPointExpression
An expression that returns a value that is a floating-point data type
(DOUBLE or REAL).

Example
Use the CHAR function to return the values for EDLEVEL (defined as smallint) as
a fixed length character string:

SELECT CHAR(EDLEVEL) FROM EMPLOYEE

An EDLEVEL of 18 would be returned as the CHAR(6) value ‘18 ’ (18 followed by
four blanks).
LENGTH

LENGTH is applied to either a character string expression or a bit string expression and returns the number of characters in the result.

Because all built-in data types are implicitly converted to strings, this function can act on all built-in data types.

Syntax
LENGTH ( { CharacterExpression | BitExpression } )

Example
-- returns 20
VALUES LENGTH('supercalifragilistic')
-- returns 1
VALUES LENGTH(X'FF')
-- returns 4
VALUES LENGTH(1234567890)
**Concatenation**

The concatenation operator, `||`, concatenates its right operand to the end of its left operand. It operates on a character or bit expression.

Because all built-in data types are implicitly converted to strings, this function can act on all built-in data types.

**Syntax**

```sql
{
    { CharacterExpression || CharacterExpression } |
    { BitExpression || BitExpression }
}
```

For character strings, if both the left and right operands are of type CHAR, the resulting type is CHAR; otherwise, it is VARCHAR. The normal blank padding/trimming rules for CHAR and VARCHAR apply to the result of this operator.

The length of the resulting string is the sum of the lengths of both operands.

For bit strings, if both the left and the right operands are of type CHAR FOR BIT DATA, the resulting type is CHAR FOR BIT DATA; otherwise, it is VARCHAR FOR BIT DATA.

**Example**

-- returns 'supercalifragilisticexbealidocious(sp?)'
VALUES 'supercalifragilistic' || 'exbealidocious' || '(sp?)'
-- returns NULL
VALUES CAST (null AS VARCHAR(7)) || 'AString'
-- returns '130asdf'
VALUES '130' || 'asdf'
**NULLIF and CASE expressions**

Use the CASE and NULLIF expressions for conditional expressions in Cloudscape.

**NULLIF expression:**

Syntax:

```
NULLIF(L, R)
```

**CASE expression:**

You can place a CASE expression anywhere an expression is allowed. It chooses an expression to evaluate based on a boolean test.

Syntax:

```
CASE WHEN BooleanExpression THEN thenExpression ELSE elseExpression END
```

ThenExpression and ElseExpression are both expressions that must be type-compatible. For built-in types, this means that the types must be the same or a built-in broadening conversion must exist between the types.

You do not need to use the CASE expression for avoiding NullPointerExceptions when a nullable column becomes a method receiver.

```
-- returns 3
VALUES CASE WHEN 1=1 THEN 3 ELSE 4 END;
```

If the value of the instance specified in an instance method invocation is null, the result of the invocation is null (SQL NULL). However, you still might need to use the CASE expression for when a nullable column is a primitive method parameter.
COUNT

COUNT is an aggregate function that counts the number of rows accessed in an expression (see “Aggregates (Set Functions)” on page 84). COUNT is allowed on all types of expressions.

Syntax

\[ \text{COUNT} \left( \left[ \text{DISTINCT} \mid \text{ALL} \right] \text{Expression} \right) \]

The DISTINCT qualifier eliminates duplicates. The ALL qualifier retains duplicates. ALL is assumed if neither ALL nor DISTINCT is specified. For example, if a column contains the values 1, 1, 1, 1, and 2, \( \text{COUNT}(\text{col}) \) returns a greater value than \( \text{COUNT}(\text{DISTINCT col}) \).

Only one DISTINCT aggregate expression per SelectExpression is allowed. For example, the following query is not allowed:

```sql
-- query not allowed
SELECT \text{COUNT}(\text{DISTINCT flying_time}), \text{SUM}(\text{DISTINCT miles})
FROM Flights
```

An Expression can contain multiple column references or expressions, but it cannot contain another aggregate or subquery. If an Expression evaluates to NULL, the aggregate is not processed for that value.

The resulting data type of COUNT is \text{BIGINT}.

Example

-- Count the number of countries in each region,
-- show only regions that have at least 2
SELECT \text{COUNT}(\text{country}), \text{region}
FROM Countries
GROUP BY region
HAVING \text{COUNT}(\text{country}) > 1

96 IBM Cloudscape: Reference Manual
COUNT(*)

COUNT(*) is an aggregate function that counts the number of rows accessed. No NULLs or duplicates are eliminated. COUNT(*) does not operate on an expression.

Syntax

COUNT(*)

The resulting data type is **BIGINT**

Example

-- Count the number of rows in the Flights table
SELECT COUNT(*)
FROM Flights
CURRENT DATE

CURRENT DATE is a synonym for "CURRENT_DATE" on page 99.
CURRENT_DATE

CURRENT_DATE returns the current date; the value returned does not change if it is executed more than once in a single statement. This means the value is fixed even if there is a long delay between fetching rows in a cursor.

Syntax

CURRENT_DATE

or, alternately

CURRENT_DATE

Example

-- find available future flights:
SELECT * FROM Flightavailability where flight_date > CURRENT_DATE;
CURRENT ISOLATION

CURRENT ISOLATION returns the current isolation level as a char(2) value of either "" (blank), "UR", "CS", "RS", or "RR".

Syntax
CURRENT ISOLATION

Example
VALUES CURRENT ISOLATION
CURRENT SCHEMA

CURRENT SCHEMA returns the schema name used to qualify unqualified database object references.

Note: CURRENT SCHEMA and CURRENT SQLID are synonyms.

These functions return a string of up to 128 characters.

Syntax
CURRENT SCHEMA
-- or, alternately:
CURRENT SQLID

Example
-- Set the name column default to the current schema:
CREATE TABLE mytable (id int, name VARCHAR(128) DEFAULT CURRENT SQLID)
-- Inserts default value of current schema value into the table:
INSERT INTO mytable(id) VALUES (1)
-- Returns the rows with the same name as the current schema:
SELECT name FROM mytable WHERE name = CURRENT SCHEMA
CURRENT TIME

CURRENT TIME is a synonym for “CURRENT_TIME” on page 103.
CURRENT_TIME

CURRENT_TIME returns the current time; the value returned does not change if it is executed more than once in a single statement. This means the value is fixed even if there is a long delay between fetching rows in a cursor.

Syntax
CURRENT_TIME

or, alternately
CURRENT TIME

Example
VALUES CURRENT_TIME
-- or, alternately:
VALUES CURRENT TIME
CURRENT TIMESTAMP

CURRENT_TIMESTAMP is a synonym for "CURRENT_TIMESTAMP" on page 105.
CURRENT_TIMESTAMP

CURRENT_TIMESTAMP returns the current timestamp; the value returned does not change if it is executed more than once in a single statement. This means the value is fixed even if there is a long delay between fetching rows in a cursor.

**Syntax**

```
CURRENT_TIMESTAMP
```

or, alternately

```
CURRENT_TIMESTAMP
```

**Example**

```
VALUES CURRENT_TIMESTAMP
-- or, alternately:
VALUES CURRENT_TIMESTAMP
```
**CURRENT_USER**

CURRENT_USER returns the authorization identifier of the current user (the name of the user passed in when the user connected to the database). If there is no current user, it returns *APP*.

USER, CURRENT_USER, and SESSION_USER are synonyms.

These functions return a string of up to 128 characters.

**Syntax**

CURRENT_USER

**Example**

VALUES CURRENT_USER
DATE

The DATE function returns a date from a value. The argument must be a date, timestamp, a positive number less than or equal to 3,652,059, a valid string representation of a date or timestamp, or a string of length 7 that is not a CLOB or LONG VARCHAR. If the argument is a string of length 7, it must represent a valid date in the form yyyy, where yyyy are digits denoting a year, and nnn are digits between 001 and 366, denoting a day of that year. The result of the function is a date. If the argument can be null, 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, timestamp, or valid string representation of a date or timestamp: The result is the date part of the value.
- 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 with a length of 7: The result is the date represented by the string.

Syntax

```
DATE ( expression )
```

Example

This example results in an internal representation of ‘1988-12-25’.

```
VALUES DATE(‘1988-12-25’)
```
DAY

The DAY function returns the day part of a value. The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR. The result of the function is a large integer. If the argument can be null, 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, timestamp, or valid string representation of a date or timestamp: The result is the day part of the value, which is an integer between 1 and 31.
- If the argument is a time 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.

**Syntax**

```
DAY ( expression )
```

**Example**

```
values day('2004-08-02');
```

The resulting value is 2.
DOUBLE

The DOUBLE function returns a floating-point number corresponding to a:
• number if the argument is a numeric expression.
• character string representation of a number if the argument is a string expression.

Syntax

Numeric to Double::
DOUBLE [PRECISION] (NumericExpression )

NumericExpression
The argument is an expression that returns a value of any built-in numeric data type.

The result of the function is a double-precision floating-point number. If the argument can be null, the result can be null; if the argument is null, the result is the null value. The result is the same number that would occur if the argument were assigned to a double-precision floating-point column or variable.

Character String to Double::
DOUBLE (StringExpression )

StringExpression
The argument can be of type CHAR or VARCHAR in the form of a numeric constant. Leading and trailing blanks in argument are ignored.

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 the same number that would occur if the string was considered a constant and assigned to a double-precision floating-point column or variable.
The HOUR function returns the hour part of a value. The argument must be a time, timestamp, or a valid character string representation of a time or timestamp that is neither a CLOB nor a LONG VARCHAR. The result of the function is a large integer. If the argument can be null, 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, timestamp, or valid string representation of a date or timestamp: The result is the hour part of the value, which is an integer between 0 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.

Syntax

```
HOUR (expression )
```

Example

Select all the classes that start in the afternoon from a table called TABLE1.

```
SELECT * FROM TABLE1
WHERE HOUR(STARTING) BETWEEN 12 AND 17
```
IDENTITY_VAL_LOCAL

Cloudscape supports the IDENTITY_VAL_LOCAL function.

Syntax:
IDENTITY_VAL_LOCAL ( )

The IDENTITY_VAL_LOCAL function is a non-deterministic function that returns the most recently assigned value for an identity column, where the assignment occurred as a result of a single row INSERT statement using a VALUES clause.

The IDENTITY_VAL_LOCAL function has no input parameters. The result is a DECIMAL (31,0), regardless of the actual data type of the corresponding identity column.

The value returned by the IDENTITY_VAL_LOCAL function is the value assigned to the identity column of the table identified in the most recent single row INSERT statement. The INSERT statement must contain a VALUES clause on a table containing an identity column. The assigned value is an identity value generated by Cloudscape. The function returns a null value when a single row INSERT statement with a VALUES clause has not been issued for a table containing an identity column.

The result of the function is not affected by the following:
- A single row INSERT statement with a VALUES clause for a table without an identity column
- A multiple row INSERT statement with a VALUES clause
- An INSERT statement with a fullselect

Examples:
ij> create table t1(c1 int generated always as identity, c2 int);
0 rows inserted/updated/deleted
ij> insert into t1(c2) values (8);
1 row inserted/updated/deleted
ij> values IDENTITY_VAL_LOCAL();
1
-------------------------------
1
1 row selected
ij> select IDENTITY_VAL_LOCAL()+1, IDENTITY_VAL_LOCAL()-1 from t1;
1
|2
-------------------------------
2 |0
1 row selected
ij> insert into t1(c2) values (IDENTITY_VAL_LOCAL());
1 row inserted/updated/deleted
ij> select * from t1;
C1 |C2
-------------------------------
1 |8
2 |1
2 rows selected
ij> values IDENTITY_VAL_LOCAL();
1
-------------------------------
2
1 row selected
ij> insert into t1(c2) values (8), (9);
2 rows inserted/updated/deleted
ij> -- multi-values insert, return value of the function should not change
values IDENTITY_VAL_LOCAL();
1
-----------------------------------
2
1 row selected
i;j> select * from t1;
C1  | C2
-----------------------------------
1   | 8
2   | 1
3   | 8
4   | 9
4 rows selected
i;j> insert into t1(c2) select c1 from t1;
4 rows inserted/updated/deleted
i;j> -- insert with sub-select, return value should not change
values IDENTITY_VAL_LOCAL();
1
-----------------------------------
2
1 row selected
i;j> select * from t1;
C1  | C2
-----------------------------------
1   | 8
2   | 1
3   | 8
4   | 9
5   | 1
6   | 2
7   | 3
8   | 4
8 rows selected
INTEGER

The INTEGER function returns an integer representation of a number, character string, date, or time in the form of an integer constant.

Syntax

\[ \text{INTEGER} \ (\text{NumericExpression} \mid \text{CharacterExpression}) \]

NumericExpression

An expression that returns a value of any built-in numeric data type. If the argument is a numeric-expression, 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 integers, an error occurs. The decimal part of the argument is truncated if present.

CharacterExpression

An expression that returns a character string value of length not greater than the maximum length of a character constant. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an SQL integer constant. The character string cannot be a long string. If the argument is a character-expression, the result is the same number that would occur if the corresponding integer constant were assigned to a large integer column or variable.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

Using the EMPLOYEE table, select a list containing salary (SALARY) divided by education level (EDLEVEL). Truncate any decimal in the calculation. The list should also contain the values used in the calculation and employee number (EMPNO). The list should be in descending order of the calculated value:

\[
\begin{align*}
\text{SELECT INTEGER} & \ (\text{SALARY} \ / \ \text{EDLEVEL}), \ \text{SALARY}, \ \text{EDLEVEL}, \ \text{EMPNO} \\
\text{FROM} & \ \text{EMPLOYEE} \\
\text{ORDER BY} & \ 1 \ \text{DESC}
\end{align*}
\]
LOCATE

If a specified substring is found within a specified search string, LOCATE returns the index at which the substring is found within the search string. If the substring is not found, LOCATE returns 0.

Syntax

```
LOCATE(CharacterExpression, CharacterExpression [, StartPosition] )
```

The second CharacterExpression is the search string and is searched from the beginning, unless `StartPosition` is specified, in which case the search begins from position there; the index starts with 1. It returns 0 if the string is not found.

The return type for LOCATE is an integer.

Example

```
-- returns 2
VALUES LOCATE('love', 'clover')
```
LCASE or LOWER

LCASE or LOWER takes a character expression as a parameter and returns a string in which all alpha characters have been converted to lowercase.

Syntax

`LCASE` or `LOWER ( CharacterExpression )`

A `CharacterExpression` is a CHAR, VARCHAR, or LONG VARCHAR data type or any built-in type that is implicitly converted to a string (except a bit expression).

If the parameter type is CHAR or LONG VARCHAR, the return type is CHAR or LONG VARCHAR. Otherwise, the return type is VARCHAR.

The length and maximum length of the returned value are the same as the length and maximum length of the parameter.

If the `CharacterExpression` evaluates to null, this function returns null.

Example

```
-- returns 'asdl#w'
VALUES LOWER('aSDI#w')

SELECT LOWER(flight_id) FROM Flights
```
LTRIM

LTRIM removes blanks from the beginning of a character string expression.

**Syntax**

LTRIM(CharacterExpression)

A CharacterExpression is a CHAR, VARCHAR, or LONG VARCHAR data type, any built-in type that is implicitly converted to a string.

LTRIM returns NULL if CharacterExpression evaluates to null.

**Example**

-- returns 'asdf '
VALUES LTRIM(' asdf ')
MAX

MAX is an aggregate function that evaluates the maximum of the expression over a set of values (see “Aggregates (Set Functions)” on page 84). MAX is allowed only on expressions that evaluate to built-in data types (including CHAR, VARCHAR, DATE, TIME, CHAR FOR BIT DATA, etc).

Syntax

MAX ( [ DISTINCT | ALL ] Expression )

The DISTINCT qualifier eliminates duplicates. The ALL qualifier retains duplicates. These qualifiers have no effect in a MAX expression. Only one DISTINCT aggregate expression per SelectExpression is allowed. For example, the following query is not allowed:

SELECT COUNT (DISTINCT flying_time), MAX (DISTINCT miles)
FROM Flights

The Expression can contain multiple column references or expressions, but it cannot contain another aggregate or subquery. It must evaluate to a built-in data type. You can therefore call methods that evaluate to built-in data types. (For example, a method that returns a java.lang.Integer or int evaluates to an INTEGER.) If an expression evaluates to NULL, the aggregate skips that value.

For CHAR, VARCHAR, and [LONG VARCHAR], the number of blank spaces at the end of the value can affect how MAX is evaluated. For example, if the values ‘z’ and ‘z ’ are both stored in a column, you cannot control which one will be returned as the maximum, because a blank space has no value.

The resulting data type is the same as the expression on which it operates (it will never overflow).

Example

-- find the latest date in the FlightAvailability table
SELECT MAX (flight_date) FROM FlightAvailability

-- find the longest flight originating from each airport,
-- but only when the longest flight is over 10 hours
SELECT MAX(flying_time), orig_airport
FROM Flights
GROUP BY orig_airport
HAVING MAX(flying_time) > 10
**MIN**

MIN is an aggregate expression that evaluates the minimum of an expression over a set of rows (see "Aggregates (Set Functions)" on page 84). MIN is allowed only on expressions that evaluate to built-in data types (including CHAR, VARCHAR, DATE, TIME, etc.).

**Syntax**

```
MIN ( [ DISTINCT | ALL ] Expression )
```

The DISTINCT and ALL qualifiers eliminate or retain duplicates, but these qualifiers have no effect in a MIN expression. Only one DISTINCT aggregate expression per `SelectExpression` is allowed. For example, the following query is not allowed:

```
SELECT COUNT (DISTINCT flying_time), MIN (DISTINCT miles)
FROM Flights
```

The expression can contain multiple column references or expressions, but it cannot contain another aggregate or subquery. It must evaluate to a built-in data type. You can therefore call methods that evaluate to built-in data types. (For example, a method that returns a `java.lang.Integer` or `int` evaluates to an INTEGER.) If an expression evaluates to NULL, the aggregate skips that value.

The type's comparison rules determine the maximum value. For CHAR, VARCHAR, and LONG VARCHAR, the number of blank spaces at the end of the value can affect the result.

The resulting data type is the same as the expression on which it operates (it will never overflow).

**Example**

```
-- NOT valid:
SELECT DISTINCT flying_time, MIN(DISTINCT miles) from Flights

-- valid:
SELECT COUNT(DISTINCT flying_time), MIN(DISTINCT miles) from Flights

-- find the earliest date:
SELECT MIN (flight_date) FROM FlightAvailability;
```
**MINUTE**

The MINUTE function returns the minute part of a value. The argument must be a time, timestamp, or a valid character string representation of a time or timestamp that is neither a CLOB nor a LONG VARCHAR. The result of the function is a large integer. If the argument can be null, 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, timestamp, or valid string representation of a date or timestamp: 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.

**Syntax**

```
MINUTE ( expression )
```

**Example**

```
-- Select all classes that do not end on a full hour:
SELECT * FROM table1 WHERE MINUTE(ending) < 60;
```
MOD

MOD returns the remainder (modulus) of argument 1 divided by argument 2. The result is negative only if argument 1 is negative.

Syntax
mod(integer_type, integer_type)

The result of the function is:
• SMALLINT if both arguments are SMALLINT.
• INTEGER if one argument is INTEGER and the other is INTEGER or SMALLINT.
• BIGINT if one integer is BIGINT and the other argument is BIGINT, INTEGER, or SMALLINT.

The result can be null; if any argument is null, the result is the null value.
MONTH

The MONTH function returns the month part of a value. The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR. The result of the function is a large integer. If the argument can be null, 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, timestamp, or valid string representation of a date or timestamp: 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.

Syntax

MONTH ( expression )

Example

Select all rows from the EMPLOYEE table for people who were born (BIRTHDATE) in DECEMBER.

SELECT * FROM EMPLOYEE
WHERE MONTH(BIRTHDATE) = 12
RTRIM

RTRIM removes blanks from the end of a character string expression.

Syntax
RTRIM(CharacterExpression)

A CharacterExpression is a CHAR, VARCHAR, or LONG VARCHAR data type, any built-in type that is implicitly converted to a string.

RTRIM returns NULL if CharacterExpression evaluates to null.

Example
-- returns 'asdf'
VALUES RTRIM(' asdf ')
-- returns 'asdf'
VALUES RTRIM('asdf ')

IBM Cloudscape: Reference Manual
SECOND

The SECOND function returns the seconds part of a value. The argument must be a time, timestamp, or a valid character string representation of a time or timestamp that is neither a CLOB nor a LONG VARCHAR. The result of the function is a large integer. If the argument can be null, 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, timestamp, or valid string representation of a date or timestamp: The result is the seconds 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 seconds part of the value, which is an integer between -99 and 99. A nonzero result has the same sign as the argument.

Syntax
SECOND ( expression )

Example
Assume that the column RECEIVED (timestamp) has an internal value equivalent to 1988-12-25-17.12.30.000000.
SECOND(RECEIVED)

Returns the value 30.
SESSION_USER

SESSION_USER returns the authorization identifier or name of the current user. If there is no current user, it returns *APP*.

*USER*, *CURRENT_USER*, and *SESSION_USER* are synonyms.

**Syntax**

```
SESSION_USER
```

**Example**

```
VALUES SESSION_USER
```
**SMALLINT**

The SMALLINT function returns a small integer representation of a number or character string in the form of a small integer constant.

**Syntax**

```sql
SMALLINT (NumericExpression | CharacterExpression)
```

**NumericExpression**

An expression that returns a value of any built-in numeric data type. If the argument is a NumericExpression, 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. The decimal part of the argument is truncated if present.

**CharacterExpression**

An expression that returns a character string value of length not greater than the maximum length of a character constant. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an SQL integer constant. However, the value of the constant must be in the range of small integers. The character string cannot be a long string. If the argument is a CharacterExpression, the result is the same number that would occur if the corresponding integer constant were assigned to a small integer column or variable.

The result of the function is a small integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.
**SQRT**

Returns the square root of a floating point number; only the built-in types [REAL, FLOAT] and [DOUBLE PRECISION] are supported. The return type for SQRT is the type of the parameter.

**Note:** To execute SQRT on other data types, you must cast them to floating point types.

**Syntax**

```
SQRT(FloatingPointExpression)
```

**Example**

```
-- throws an exception if any row stores a negative number:
VALUES SQRT(3421E+09)

-- returns the square root of an INTEGER after casting it as a floating point data type:
SELECT SQRT(myDoubleColumn) FROM MyTable

VALUES SQRT (CAST(25 AS FLOAT));
```
**SUBSTR**

The SUBSTR function acts on a character string expression or a bit string expression. The type of the result is a `VARCHAR` in the first case and `VARCHAR FOR BIT DATA` in the second case. The length of the result is the maximum length of the source type.

**Syntax**

```
SUBSTR({ CharacterExpression },
    StartPosition [, LengthOfString ] )
```

`startPosition` and the optional `lengthOfString` are both integer expressions. (The first character or bit has a `startPosition` of 1; if you specify 0, Cloudscape assumes that you mean 1.)

A `characterExpression` is a CHAR, VARCHAR, or LONG VARCHAR data type or any built-in type that is implicitly converted to a string (except a bit expression).

For character expressions, both `startPosition` and `lengthOfString` refer to characters. For bit expressions, both `startPosition` and `lengthOfString` refer to bits.

SUBSTR returns NULL if `lengthOfString` is specified and it is less than zero.

If `startPosition` is positive, it refers to position from the start of the source expression (counting the first character as 1). If `startPosition` is negative, it is the position from the end of the source.

If `lengthOfString` is not specified, SUBSTR returns the substring of the expression from the `startPosition` to the end of the source expression. If `lengthOfString` is specified, SUBSTR returns a VARCHAR or VARBIT of length `lengthOfString` starting at the `startPosition`. 
SUM

SUM is an aggregate expression that evaluates the sum of the expression over a set of rows (see “Aggregates (Set Functions)” on page 84). SUM is allowed only on expressions that evaluate to numeric data types.

Syntax

\[
\text{SUM ( [ DISTINCT | ALL ] Expression )}
\]

The DISTINCT and ALL qualifiers eliminate or retain duplicates. ALL is assumed if neither ALL nor DISTINCT is specified. For example, if a column contains the values 1, 1, 1, 1, and 2, \( \text{SUM(col)} \) returns a greater value than \( \text{SUM(DISTINCT col)} \).

Only one DISTINCT aggregate expression per SelectExpression is allowed. For example, the following query is not allowed:

```sql
SELECT AVG (DISTINCT flying_time), SUM (DISTINCT miles)
FROM Flights
```

The Expression can contain multiple column references or expressions, but it cannot contain another aggregate or subquery. It must evaluate to a built-in numeric data type. If an expression evaluates to NULL, the aggregate skips that value.

The resulting data type is the same as the expression on which it operates (it might overflow).

Example

-- find all economy seats available:
SELECT SUM (economy_seats) FROM Airlines;

-- use SUM on multiple column references
-- (find the total number of all seats purchased):
SELECT SUM (economy_seats_taken + business_seats_taken + firstclass_seats_taken) as seats_taken FROM FLIGHTAVAILABILITY;
**TIME**

The TIME function returns a time from a value. The argument must be a time, timestamp, or a valid string representation of a time or timestamp that is not a CLOB or LONG VARCHAR. The result of the function is a time. If the argument can be null, 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 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 represented by the string.

**Syntax**

```
TIME ( expression )
```

**Example**

```
values time(current_timestamp)
```

If the current time is 5:03 PM, the value returned is 17:03:00.
TIMESTAMP

The TIMESTAMP function returns a timestamp from a value or a pair of values.

The rules for the arguments depend on whether the second argument is specified:

- If only one argument is specified: It must be a timestamp, a valid string representation of a timestamp, or a string of length 14 that is not a CLOB or LONG VARCHAR. A string of length 14 must be a string of digits 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 both arguments are specified: The first argument must be a date or a valid string representation of a date and the second argument must be a time or a valid string representation of a time.

The other rules depend on whether the second argument is specified:

- If both arguments are specified: The result is a timestamp with the date specified by the first argument and the time specified by the second argument. The microsecond part of the timestamp is zero.

- If only one argument is specified and it is a timestamp: The result is that timestamp.

- If only one argument is specified and it is a string: The result is the timestamp represented by that string. If the argument is a string of length 14, the timestamp has a microsecond part of zero.

Syntax

TIMESTAMP ( expression [, expression ] )

Example

Assume the column START_DATE (date) has a value equivalent to 1988-12-25, and the column START_TIME (time) has a value equivalent to 17.12.30.

TIMESTAMP(START_DATE, START_TIME)

Returns the value ‘1988-12-25-17.12.30.000000’.
**UCASE or UPPER**

UCASE or UPPER takes a character expression as a parameter and returns a string in which all alpha characters have been converted to uppercase.

**Syntax**

```sql
UCASE or UPPER ( CharacterExpression )
```

If the parameter type is CHAR, the return type is CHAR. Otherwise, the return type is VARCHAR.

**Note:** UPPER and LOWER follow the database locale. See "territory=ll_CC" on page 240 for more information about specifying locale.

The length and maximum length of the returned value are the same as the length and maximum length of the parameter.

**Example**

```sql
-- returns 'ASD1#W'
VALUES UPPER('aSD1#w')
```
USER

USER returns the authorization identifier or name of the current user. If there is no current user, it returns APP.

USER, CURRENT_USER, and SESSION_USER are synonyms.

Syntax

```
USER
```

Example

```
VALUES USER
```
**VARCHAR**

The VARCHAR function returns a varying-length character string representation of a character string.

**Syntax**

**Character to Varchar:**

\[
\text{VARCHAR} \ (\text{CharacterStringExpression} \ )
\]

**CharacterStringExpression**

An expression whose value must be of a character-string data type with a maximum length of 32,672 bytes.

**Datetime to Varchar:**

\[
\text{VARCHAR} \ (\text{DatetimeExpression} \ )
\]

**DatetimeExpression**

An expression whose value must be of a date, time, or timestamp data type.

**Example**

Using the EMPLOYEE table, select the job description (JOB defined as CHAR(8)) for Dolores Quintana as a VARCHAR equivalent:

\[
\text{SELECT VARCHAR(JOB) FROM EMPLOYEE WHERE LASTNAME = 'QUINTANA'}
\]
YEAR

The YEAR function returns the year part of a value. The argument must be a date, timestamp, or a valid character string representation of a date or timestamp. The result of the function is a large integer. If the argument can be null, 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, timestamp, or valid string representation of a date or timestamp: 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 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.

Syntax

YEAR ( expression )

Example

Select all the projects in the PROJECT table that are scheduled to start (PRSTDATE) and end (PRENDATE) in the same calendar year.

SELECT * FROM PROJECT
WHERE YEAR(PRSTDATE) = YEAR(PRENDATE)
Built-in system functions

This section describes the different built-in system functions available with Cloudscape.

- "SYSCS_UTIL.SYSCS_CHECK_TABLE"
- "SYSCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS"
- "SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY"

SYCS_UTIL.SYSCS_CHECK_TABLE

The SYCS_UTIL.SYSCS_CHECK_TABLE function checks the specified table, ensuring that all of its indexes are consistent with the base table. When tables are consistent, the method returns a SMALLINT with value 1. If the tables are inconsistent, the function will throw an exception.

Syntax:

```
SMALLINT SYCS_UTIL.SYSCS_CHECK_TABLE(IN SCHEMANAME VARCHAR(128),
IN TABLENAME VARCHAR(128))
```

An error will occur if either SCHEMANAME or TABLENAME are null.

Example:

```
VALUES SYCS_UTIL.SYSCS_CHECK_TABLE('SALES', 'ORDERS');
```

SYCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS

The SYCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS function returns a VARCHAR(32762) value representing the query execution plan and run time statistics for a java.sql.ResultSet. A query execution plan is a tree of execution nodes. There are a number of possible node types. Statistics are accumulated during execution at each node. The types of statistics include the amount of time spent in specific operations, the number of rows passed to the node by its children, and the number of rows returned by the node to its parent. (The exact statistics are specific to each node type.) SYCS_UTIL.SYCS_GET_RUNTIMESTATISTICS is most meaningful for DML statements such as SELECT, INSERT, DELETE and UPDATE.

Syntax:

```
VARCHAR(32762) SYCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS()
```

Example:

```
VALUES SYCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS()
```

SYCS_UTIL.SYSCS_GET_DATABASE_PROPERTY

The SYCS_UTIL.SYSCS_GET_DATABASE_PROPERTY function fetches the value of a property specified by KEY of the database on the current connection.

Syntax:

```
VARCHAR(32762) SYCS_UTIL.SYSCS_GET_DATABASE_PROPERTY(IN KEY VARCHAR(128))
```

An error will be returned if KEY is null.

Example:

```
VALUES SYCS_UTIL.SYSCS_GET_DATABASE_PROPERTY('key_value_string');
```
Built-in system procedures

The following built-in procedures are not compatible with SQL syntax used by IBM® DB2 Universal Database. These procedures can only be used with Cloudscape.

- "SYSCS_UTIL.SYSCS_COMPRESS_TABLE"
- "SYSCS_UTIL.SYSCS_SET_RUNTIMESTATISTICS” on page 137
- "SYSCS_UTIL.SYSCS_SET_STATISTICS_TIMING” on page 138
- "SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY” on page 138
- "SYSCS_UTIL.SYSCS_FREEZE_DATABASE” on page 139
- "SYSCS_UTIL.SYSCS_UNFREEZE_DATABASE” on page 139
- "SYSCS_UTIL.SYSCS_CHECKPOINT_DATABASE” on page 139
- "SYSCS_UTIL.SYSCS_BACKUP_DATABASE” on page 140
- "SYSCS_UTIL.SYSCS_EXPORT_TABLE” on page 140
- "SYSCS_UTIL.SYSCS_EXPORT_QUERY” on page 141
- "SYSCS_UTIL.SYSCS_IMPORT_TABLE” on page 142
- "SYSCS_UTIL.SYSCS_IMPORT_DATA” on page 143

SYCS_UTIL.SYSCS_COMPRESS_TABLE

Use the SYCS_UTIL.SYSCS_COMPRESS_TABLE system procedure to reclaim unused, allocated space in a table and its indexes. Typically, unused allocated space exists when a large amount of data is deleted from a table, or indexes are updated. By default, Cloudscape does not return unused space to the operating system. For example, once a page has been allocated to a table or index, it is not automatically returned to the operating system until the table or index is destroyed. SYCS_UTIL.SYSCS_COMPRESS_TABLE allows you to return unused space to the operating system.

Syntax:
SYCS_UTIL.SYSCS_COMPRESS_TABLE (IN SCHEMANAME VARCHAR(128), IN TABLENAME VARCHAR(128), IN SEQUENTIAL SMALLINT)

SCHEMANAME
An input argument of type VARCHAR(128) that specifies the schema of the table. Passing a null will result in an error.

TABLENAME
An input argument of type VARCHAR(128) that specifies the table name of the table. The string must exactly match the case of the table name, and the argument of "Fred" will be passed to SQL as the delimited identifier ‘Fred’. Passing a null will result in an error.

SEQUENTIAL
A non-zero input argument of type SMALLINT will force the operation to run in sequential mode, while an argument of 0 will force the operation not to run in sequential mode. Passing a null will result in an error.

SQL example:

To compress a table called CUSTOMER in a schema called US, using the SEQUENTIAL option:
call SYCS_UTIL.SYSCS_COMPRESS_TABLE('US', 'CUSTOMER', 1)

Java example:
To compress a table called CUSTOMER in a schema called US, using the SEQUENTIAL option:

```java
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_COMPRESS_TABLE(?, ?, ?)\n);
cs.setString(1, "US");
cs.setString(2, "CUSTOMER");
cs.setShort(3, (short) 1);
cs.execute();
```

If the SEQUENTIAL parameter is not specified, Cloudscape rebuilds all indexes concurrently with the base table. If you do not specify the SEQUENTIAL argument, this procedure can be memory-intensive and use a lot of temporary disk space (an amount equal to approximately two times the used space plus the unused, allocated space). This is because Cloudscape compresses the table by copying active rows to newly allocated space (as opposed to shuffling and truncating the existing space). The extra space used is returned to the operating system on COMMIT.

When SEQUENTIAL is specified, Cloudscape compresses the base table and then compresses each index sequentially. Using SEQUENTIAL uses less memory and disk space, but is more time-intensive. Use the SEQUENTIAL argument to reduce memory and disk space usage.

SYSCS_UTIL.SYSCS_COMPRESS_TABLE cannot release any permanent disk space back to the operating system until a COMMIT is issued. This means that the space occupied by both the base table and its indexes cannot be released back to the operating system until a COMMIT is issued. (Only the disk space that is temporarily claimed by an external sort can be returned to the operating system prior to a COMMIT.) We recommended you issue the SYSCS_UTIL.SYSCS_COMPRESS_TABLE procedure in auto-commit mode.

**Note:** This procedure acquires an exclusive table lock on the table being compressed. All statement plans dependent on the table or its indexes are invalidated. For information on identifying unused space, see the *IBM Cloudscape Server and Administration Guide*.

**SYSCS_UTIL.SYSCS_SET_RUNTIMESTATISTICS**

The SYSCS_UTIL.SYSCS_SET_RUNTIMESTATISTICS() system procedure turns a connection’s runtime statistics on or off. By default, the runtime statistics are turned off. When the runtimestatistics attribute is turned on, Cloudscape maintains information about the execution plan for each statement executed within the connection (except for COMMIT) until the attribute is turned off. To turn the runtimestatistics attribute off, call the procedure with an argument of zero. To turn the runtimestatistics on, call the procedure with any non-zero argument.

For statements that do not return rows, the object is created when all internal processing has completed before returning to the client program. For statements that return rows, the object is created when the first next() call returns 0 rows or if a close() call is encountered, whichever comes first.

**Syntax:**

SYSCS_UTIL.SYSCS_SET_RUNTIMESTATISTICS(IN SMALLINT ENABLE)

**Example:**
-- establish a connection
-- turn on RUNITMESTATISTIC for connection:
CALL SYSCS_UTIL.SYSCS_SET_RUNTIMESTATISTICS(1);
-- execute complex query here
-- step through the result sets
-- access runtime statistics information:
CALL SYSCS_UTIL.SYSCS_SET_RUNTIMESTATISTICS(0);

SYSCS_UTIL.SYSCS_SET_STATISTICS_TIMING

Statistics timing is an attribute associated with a connection that you turn on and off by using the SYSCS_UTIL.SYSCS_SET_STATISTICS_TIMING system procedure. Statistics timing is turned off by default. Turn statistics timing on only when the runtimestatistics attribute is already on. Turning statistics timing on when the runtimestatistics attribute is off has no effect.

Turn statistics timing on by calling this procedure with a non-zero argument. Turn statistics timing off by calling the procedure with a zero argument.

When statistics timing is turned on, Cloudscape tracks the timings of various aspects of the execution of a statement. This information is included in the information returned by the "SYSCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS" on page 135 system function. When statistics timing is turned off, the "SYSCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS" on page 135 system function shows all timing values as zero.

Syntax:
SYSCS_UTIL.SYSCS_SET_STATISTICS_TIMING(IN SMALLINT ENABLE)

Example:

To turn the runtimestatistics attribute and then the statistics timing attribute on:
CALL SYSCS_UTIL.SYSCS_SET_RUNTIMESTATISTICS(1);
CALL SYSCS_UTIL.SYSCS_SET_STATISTICS_TIMING(1);

SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY

Use the SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY system procedure to set or delete the value of a property of the database on the current connection.

If "VALUE" is not null, then the property with key value "KEY" is set to "VALUE". If "VALUE" is null, then the property with key value "KEY" is deleted from the database property set.

Syntax:
SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY(IN KEY VARCHAR(128),
IN VALUE VARCHAR(32672))

This procedure does not return any results.

JDBC Example:

Set the derby.locks.deadlockTimeout property to a value of 10:

```
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(?, ?)");
cc.setString(1, "derby.locks.deadlockTimeout");
cs.setString(2, "10");
cs.execute();
cs.close();
```
SQL Example:

Set the derby.locks.deadlockTimeout property to a value of 10:

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
('derby.locks.deadlockTimeout', '10');
```

**SYSCS_UTIL.SYSCS_FREEZE_DATABASE**

The **SYSCS_UTIL.SYSCS_FREEZE_DATABASE** system procedure temporarily freezes the database for backup.

**Syntax:**

```
SYSCS_UTIL.SYSCS_FREEZE_DATABASE()
```

No result set is returned by this procedure.

**Example:**

```
String backupdirectory = "c:/mybackups/" + JCalendar.getToday();
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_FREEZE_DATABASE()");
   cs.execute();
   cs.close();
// user supplied code to take full backup of "backupdirectory"
// now unfreeze the database once backup has completed:
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_UNFREEZE_DATABASE()");
   cs.execute();
   cs.close();
```

**SYSCS_UTIL.SYSCS_UNFREEZE_DATABASE**

The **SYSCS_UTIL.SYSCS_UNFREEZE_DATABASE** system procedure unfreezes a database after backup.

**Syntax:**

```
SYSCS_UTIL.SYSCS_UNFREEZE_DATABASE()
```

No result set is returned by this procedure.

**Example:**

```
String backupdirectory = "c:/mybackups/" + JCalendar.getToday();
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_FREEZE_DATABASE()");
   cs.execute();
   cs.close();
// user supplied code to take full backup of "backupdirectory"
// now unfreeze the database once backup has completed:
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_UNFREEZE_DATABASE()");
   cs.execute();
   cs.close();
```

**SYSCS_UTIL.SYSCS_CHECKPOINT_DATABASE**

The **SYSCS_UTIL.SYSCS_CHECKPOINT_DATABASE** system procedure checkpoints the database by flushing all cached data to disk.

**Syntax:**

```
SYSCS_UTIL.SYSCS_CHECKPOINT_DATABASE()
```
No result is returned by this procedure.

**JDBC Example:**
```java
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_CHECKPOINT_DATABASE()")
cs.execute();
cs.close();
```

**SQL Example:**
```sql
CALL SYSCS_UTIL.SYSCS_CHECKPOINT_DATABASE();
```

### SYSCS_UTIL.SYSCS_BACKUP_DATABASE

The SYSCS_UTIL.SYSCS_BACKUP_DATABASE system procedure backs up the database to a specified backup directory.

**Syntax:**
```
SYSCS_UTIL.SYSCS_BACKUP_DATABASE(IN BACKUPDIR VARCHAR())
```

No result is returned from the procedure.

**backupDir**

An input argument of type VARCHAR(32672) that specifies the full system path to the database directory to be backed up.

**JDBC Example:**
```java
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_BACKUP_DATABASE(?)")
cs.setString(1, "c:/backupdir");
cs.execute();
cs.close();
```

**SQL Example:**
```sql
CALL SYSCS_UTIL.SYSCS_BACKUP_DATABASE('c:/backupdir');
```

### SYSCS_UTIL.SYSCS_EXPORT_TABLE

The SYSCS_UTIL.SYSCS_EXPORT_TABLE system procedure exports all of the data from a table to an operating system file in a delimited data file format.

**Syntax:**
```
SYSCS_UTIL.SYSCS_EXPORT_TABLE (IN SCHEMANAME VARCHAR(128),
IN TABLENAME VARCHAR(128),
IN FILENAME VARCHAR(32672),
IN COLUMNDELIMITER CHAR(1),
IN CHARACTERDELIMITER CHAR(1),
IN CODESET VARCHAR(128))
```

No result is returned from the procedure.

**SCHEMANAME**

An input argument of type VARCHAR(128) that specifies the schema name of the table. Passing a NULL value will use the default schema name.
TABLENAME
An input argument of type VARCHAR(128) that specifies the name of the table/view from which the data is to be exported. Passing a null will result in an error.

FILENAME
An input argument of type VARCHAR(32672) that specifies the name of the file to which data is to be exported. If the complete path to the file is not specified, the export procedure uses the current directory and the default drive as the destination. If the name of a file that already exists is specified, the export procedure overwrites the contents of the file; it does not append the information. Passing a null will result in an error.

COLUMNDELIMITER
An input argument of type CHAR(1) that specifies a column delimiter. The specified character is used in place of a comma to signal the end of a column. Passing a NULL value will use the default value; the default value is a comma (,).

CHARACTERDELIMITER
An input argument of type CHAR(1) that specifies a character delimiter. The specified character is used in place of double quotation marks to enclose a character string. Passing a NULL value will use the default value; the default value is a double quotation mark (").

CODESET
An input argument of type VARCHAR(128) that specifies the code set of the data in the exported file. The name of the code set should be one of the Java-supported character encodings. Data is converted from the database code set to the specified code set before writing to the file. Passing a NULL value will write the data in the same code set as the JVM in which it is being executed.

Example:
The following example shows how to export information from the STAFF table in a SAMPLE database to the myfile.del file.
CALL SYSCS_UTIL.SYSCS_EXPORT_TABLE (null, 'staff', 'myfile.del', null, null, null);

For more information on exporting, see the IBM Cloudscape Tools and Utilities Guide.

SYCS_UTIL.SYSCS_EXPORT_QUERY
The SYCS_UTIL.SYSCS_EXPORT_QUERY system procedure exports the results of a SELECT statement to an operating system file in a delimited data file format.

Syntax:
SYCS_UTIL.SYSCS_EXPORT_QUERY(IN SELECTSTATEMENT VARCHAR(32672),
IN FILENAME VARCHAR(32672), IN COLUMNDELIMITER CHAR(1),
IN CHARACTERDELIMITER CHAR(1), IN CODESET VARCHAR(128))

No result is returned from the procedure.

SELECTSTATEMENT
An input argument of type VARCHAR(32672) that specifies the select statement (query) that will return the data to be exported. Passing a NULL value will result in an error.
FILENAME
An input argument of type VARCHAR(32672) that specifies the name of
the file to which data is to be exported. If the complete path to the file is
not specified, the export procedure uses the current directory and the
default drive as the destination. If the name of a file that already exists is
specified, the export procedure overwrites the contents of the file; it does
not append the information. Passing a null will result in an error.

COLUMNDELMITER
An input argument of type CHAR(1) that specifies a column delimiter. The
specified character is used in place of a comma to signal the end of a
column. Passing a NULL value will use the default value; the default value
is a comma (,).

CHARACTERDELMITER
An input argument of type CHAR(1) that specifies a character delimiter.
The specified character is used in place of double quotation marks to
enclose a character string. Passing a NULL value will use the default value;
the default value is a double quotation mark (").

CODESET
An input argument of type VARCHAR(128) that specifies the code set of
the data in the exported file. The name of the code set should be one of
the Java-supported character encodings. Data is converted from the database
code set to the specified code set before writing to the file. Passing a NULL
value will write the data in the same code set as the JVM in which it is
being executed.

Example:

The following example shows how to export the information about employees in
Department 20 from the STAFF table in the SAMPLE database to the myfile.del
file.
CALL SYSCS_UTIL.SYSCS_EXPORT_QUERY('select * from staff where dept =20',
'c:/output/awards.del', null, null, null);

For more information on exporting, see the IBM Cloudscape Tools and Utilities Guide.

SYSCS_UTIL.SYSCS_IMPORT_TABLE
The SYSCS_UTIL.SYSCS_IMPORT_TABLE system procedure imports data from an input
file into all of the columns of a table. If the table receiving the imported data
already contains data, you can either replace or append to the existing data.

Syntax:
SYSCS_UTIL.SYSCS_IMPORT_TABLE (IN SCHEMANAME VARCHAR(128),
IN TABLENAME VARCHAR(128), IN FILENAME VARCHAR(32672),
IN COLUMNDELMITER CHAR(1), IN CHARACTERDELMITER CHAR(1),
IN CODESET VARCHAR(128), IN REPLACE SMALLINT)

No result is returned from the procedure.

SCHEMANAME
An input argument of type VARCHAR(128) that specifies the schema of
the table. Passing a NULL value will use the default schema name.

TABLENAME
An input argument of type VARCHAR (128) that specifies the table name
of the table into which the data is to be imported. This table cannot be a system table or a declared temporary table. Passing a null will result in an error.

FILENAME
An input argument of type VARCHAR(32672) that specifies the file that contains the data to be imported. If you do not specify a path, the current working directory is used. Passing a NULL value will result in an error.

COLUMNDELIMITER
An input argument of type CHAR(1) that specifies a column delimiter. The specified character is used in place of a comma to signal the end of a column. Passing a NULL value will use the default value; the default value is a comma (,).

CHARACTERDELIMITER
An input argument of type CHAR(1) that specifies a character delimiter. The specified character is used in place of double quotation marks to enclose a character string. Passing a NULL value will use the default value; the default value is a double quotation mark (").

CODESET
An input argument of type VARCHAR(128) that specifies the code set of the data in the input file. The name of the code set should be one of the Java-supported character encodings. Data is converted from the specified code set to the database code set (utf-8). Passing a NULL value will interpret the data file in the same code set as the JVM in which it is being executed.

REPLACE
A input argument of type SMALLINT. A non-zero value will run in REPLACE mode, while a value of zero will run in INSERT mode. REPLACE mode deletes all existing data from the table by truncating the data object, and inserts the imported data. The table definition and the index definitions are not changed. INSERT mode adds the imported data to the table without changing the existing table data. Passing a NULL will result in an error.

Example:

The following example imports data into the staff table from a delimited data file called myfile.del with the percentage character (%) as the string delimiter, and a semicolon (;) as the column delimiter:

CALL SYSCS_UTIL.SYSCS_IMPORT_TABLE
(null, 'staff', 'c:/output/myfile.del', ';', '%', null,0);

For more information on importing, see the IBM Cloudscape Tools and Utilities Guide.

SYCS_UTIL.SYSCS_IMPORT_DATA
The SYCS_UTIL.SYSCS_IMPORT_DATA system procedure imports data to a subset of columns in a table. You choose the subset of columns by specifying insert columns. This procedure is also used to import a subset of column data from a file by specifying column indexes.

Syntax:
No result is returned from the procedure.

**SCHEMANAME**

An input argument of type VARCHAR(128) that specifies the schema of the table. Passing a NULL value will use the default schema name.

**TABLENAME**

An input argument of type VARCHAR (128) that specifies the table name of the table into which the data is to be imported. This table cannot be a system table or a declared temporary table. Passing a null will result in an error.

**INSERTCOLUMNS**

An input argument of type VARCHAR (128) that specifies the column names (separated by commas) of the table into which the data is to be imported. Passing a NULL value will import the data into all of the columns of the table.

**COLUMNINDEXES**

An input argument of type VARCHAR (128) that specifies the indexes (numbered from 1 and separated by commas) of the input data fields to be imported. Passing a NULL value will use all of the input data fields in the file.

**FILENAME**

An input argument of type VARCHAR(32672) that specifies the file that contains the data to be imported. If you do not specify a path, the current working directory is used. Passing a NULL value will result in an error.

**COLUMNDELEIMITER**

An input argument of type CHAR(1) that specifies a column delimiter. The specified character is used in place of a comma to signal the end of a column. Passing a NULL value will use the default value; the default value is a comma (,).

**CHARACTERDELEIMITER**

An input argument of type CHAR(1) that specifies a character delimiter. The specified character is used in place of double quotation marks to enclose a character string. Passing a NULL value will use the default value; the default value is a double quotation mark (").

**CODESET**

An input argument of type VARCHAR(128) that specifies the code set of the data in the input file. The name of the code set should be one of the Java-supported character encodings. Data is converted from the specified code set to the database code set (utf-8). Passing a NULL value will interpret the data file in the same code set as the JVM in which it is being executed.

**REPLACE**

A input argument of type SMALLINT. A non-zero value will run in REPLACE mode, while a value of zero will run in INSERT mode. REPLACE mode deletes all existing data from the table by truncating the data object, and inserts the imported data. The table definition and the index definitions are not changed. You can only use the REPLACE mode if
the table exists. INSERT mode adds the imported data to the table without changing the existing table data. Passing a NULL will result in an error.

Example:

The following example imports some of the data fields from a delimited data file called data.del into the staff table:

```
CALL SYSCS_UTIL.SYSCS_IMPORT_DATA
(NULL, 'staff', null, '1,3,4', 'data.del', null, null, null,0)
```

For more information on importing, see the IBM Cloudscape Tools and Utilities Guide.

### Data Types

- “Built-In Type Overview”
- “Numeric Types” on page 146
- “Data type assignments and comparison, sorting, and ordering” on page 149
- Appendix B, “Supported territories,” on page 255

### Built-In Type Overview

The SQL type system is used by the language compiler to determine the compile-time type of an expression and by the language execution system to determine the runtime type of an expression, which can be a subtype or implementation of the compile-time type.

Each type has associated with it values of that type. In addition, values in the database or resulting from expressions can be NULL, which means the value is missing or unknown. Although there are some places where the keyword NULL can be explicitly used, it is not in itself a value, because it needs to have a type associated with it.

The syntax presented in this section is the syntax you use when specifying a column’s data type in a CREATE TABLE statement.

Version 10 supports the following data types:

- “BIGINT” on page 153
- “CHAR FOR BIT DATA” on page 156
- “VARCHAR FOR BIT DATA” on page 173
- “BLOB” on page 154
- “CHAR” on page 155
- “CLOB” on page 157
- “DATE” on page 159
- “DECIMAL” on page 160
- “DOUBLE PRECISION” on page 162
- “FLOAT” on page 163
- “INTEGER” on page 164
- “LONG VARCHAR FOR BIT DATA” on page 166
- “LONG VARCHAR” on page 165
- “NUMERIC” on page 167
- “REAL” on page 168
- “SMALLINT” on page 169
Numeric Types

- “Numeric Type Overview”
- “Numeric Type Promotion in Expressions”
- “Storing Values of One Numeric Data Type in Columns of Another Numeric Data Type” on page 147
- “Scale for Decimal Arithmetic” on page 147

Numeric Type Overview

Numeric types include the following types, which provide storage of varying sizes:

- Integer numerics
  - SMALLINT (2 bytes)
  - INTEGER (4 bytes)
  - BIGINT (8 bytes)
- Approximate or floating-point numerics
  - REAL (4 bytes)
  - DOUBLE PRECISION (8 bytes)
  - FLOAT (an alias for DOUBLE PRECISION or REAL)
- Exact numeric
  - DECIMAL (storage based on precision)
  - NUMERIC (an alias for DECIMAL)

Numeric Type Promotion in Expressions

In expressions that use only integer types, Cloudscape promotes the type of the result to at least INTEGER. In expressions that mix integer with non-integer types, Cloudscape promotes the result of the expression to the highest type in the expression. Table 3 shows the promotion of data types in expressions.

<table>
<thead>
<tr>
<th>Largest Type That Appears in Expression</th>
<th>Resulting Type of Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOUBLE PRECISION</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>REAL</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>

For example:

-- returns a double precision
VALUES 1 + 1.0e0
-- returns a decimal
VALUES 1 + 1.0
-- returns an integer
VALUES CAST (1 AS INT) + CAST (1 AS INT)
Storing Values of One Numeric Data Type in Columns of Another Numeric Data Type

An attempt to put a floating-point type of a larger storage size into a location of a smaller size fails only if the value cannot be stored in the smaller-size location. For example:

```sql
CREATE TABLE mytable (r REAL, d DOUBLE PRECISION);
INSERT INTO mytable (r, d) VALUES (3.4028236E38, 3.4028235E38);
ERROR X0X41: The number '3.4028236E38' is outside the range for the data type REAL.
```

You can store a floating point type in an INTEGER column; the fractional part of the number is truncated. For example:

```sql
INSERT INTO mytable(integer_column) VALUES (1.09e0);
1 row inserted/updated/deleted
SELECT integer_column FROM mytable;
```

Integer types can always be placed successfully in approximate numeric values, although with the possible loss of some precision.

Integers can be stored in decimals if the DECIMAL precision is large enough for the value. For example:

```sql
INSERT INTO mytable(decimal_column) VALUES (55555555556666666666);
ERROR X0Y21: The number '55555555556666666666' is outside the range of the target DECIMAL/NUMERIC(5,2) datatype.
```

An attempt to put an integer value of a larger storage size into a location of a smaller size fails if the value cannot be stored in the smaller-size location. For example:

```sql
INSERT INTO mytable (int_column) VALUES 2147483648;
ERROR 22003: The resulting value is outside the range for the data type INTEGER.
```

Note: When truncating trailing digits from a NUMERIC value, Cloudscape rounds down.

**Scale for Decimal Arithmetic**

SQL statements can involve arithmetic expressions that use decimal data types of different *precisions* (the total number of digits, both to the left and to the right of the decimal point) and *scales* (the number of digits of the fractional component). The precision and scale of the resulting decimal type depend on the precision and scale of the operands.

Given an arithmetic expression that involves two decimal operands:

- \( lp \) stands for the precision of the left operand
- \( rp \) stands for the precision of the right operand
- \( ls \) stands for the scale of the left operand
- \( rs \) stands for the scale of the right operand

Use the following formulas to determine the scale of the resulting data type for the following kinds of arithmetical expressions:
• *multiplication*
  \[ \text{ls} + \text{rs} \]
• *division*
  \[ 31 - \text{lp} + \text{ls} - \text{rs} \]
• *\text{AVG}()*
  \[ \max(\max(\text{ls}, \text{rs}), 4) \]
• *all others*
  \[ \max(\text{ls}, \text{rs}) \]

For example, the scale of the resulting data type of the following expression is 27:
\[ 11.0/1111.33 \]
// 31 - 3 + 1 - 2 = 27

Use the following formulas to determine the precision of the resulting data type for the following kinds of arithmetical expressions:

• *multiplication*
  \[ \text{lp} + \text{rp} \]

• *addition*
  \[ 2 \times (\text{p} - \text{s}) + \text{s} \]

• *division*
  \[ \text{lp} - \text{ls} + \text{rp} + \max(\text{ls} + \text{rp} - \text{rs} + 1, 4) \]

• *all others*
  \[ \max(\text{lp} - \text{ls}, \text{rp} - \text{rs}) + 1 + \max(\text{ls}, \text{rs}) \]
Data type assignments and comparison, sorting, and ordering

Table 4. Assignments allowed by Cloudscape. This table displays valid assignments between data types in Cloudscape. A "Y" indicates that the assignment is valid.

<table>
<thead>
<tr>
<th>Types</th>
<th>SMALL INT</th>
<th>INTEGER</th>
<th>BIGINT</th>
<th>DECIMAL</th>
<th>REAL</th>
<th>DOUBLE</th>
<th>FLOAT</th>
<th>CHAR</th>
<th>VARCHAR</th>
<th>CHAR FOR BIT DATA</th>
<th>VARCHAR FOR BIT DATA</th>
<th>LONG VARCHAR FOR BIT DATA</th>
<th>CLOB</th>
<th>BLOB</th>
<th>DATE</th>
<th>TIMESTAMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALL INT</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>INTEGER</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>BIGINT</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>REAL</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>FLOAT</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>CHAR</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>LONG VARCHAR</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>CHAR FOR BIT DATA</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>VARCHAR FOR BIT DATA</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>LONG VARCHAR FOR BIT DATA</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>CLOB</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>BLOB</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>DATE</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Chapter 1. SQL Language Reference  149
Table 4. Assignments allowed by Cloudscape (continued). This table displays valid assignments between data types in Cloudscape. A "Y" indicates that the assignment is valid.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>TIME STAMP</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
Table 5. Comparisons allowed by Cloudscape. This table displays valid comparisons between data types in Cloudscape. A “Y” indicates that the comparison is allowed.

<table>
<thead>
<tr>
<th>Types</th>
<th>SMALL INT</th>
<th>INTEGER</th>
<th>BIGINT</th>
<th>DECIMAL</th>
<th>REAL</th>
<th>DOUBLE</th>
<th>FLOAT</th>
<th>CHAR</th>
<th>VARCHAR FOR BIT DATA</th>
<th>LONG VARCHAR FOR BIT DATA</th>
<th>CHAR FOR BIT DATA</th>
<th>VARCHAR FOR BIT DATA</th>
<th>LONG VARCHAR FOR BIT DATA</th>
<th>CLOB</th>
<th>DATE</th>
<th>TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALL INT</td>
<td>Y Y Y Y Y Y — — — — — — — — —</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>Y Y Y Y Y Y — — — — — — — — —</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BIGINT</td>
<td>Y Y Y Y Y Y — — — — — — — — —</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DECIMAL</td>
<td>Y Y Y Y Y Y — — — — — — — — —</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REAL</td>
<td>Y Y Y Y Y Y — — — — — — — — —</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOUBLE</td>
<td>Y Y Y Y Y Y — — — — — — — — —</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLOAT</td>
<td>Y Y Y Y Y Y — — — — — — — — —</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHAR</td>
<td>— — — — — — Y Y — — — — — — Y Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VARCHAR</td>
<td>— — — — — — Y Y — — — — — — Y Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LONG VARCHAR</td>
<td>— — — — — — — — — — — — — — — —</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHAR FOR BIT DATA</td>
<td>— — — — — — — — Y Y — — — — — —</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VARCHAR FOR BIT DATA</td>
<td>— — — — — — — — Y Y — — — — — —</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LONG VARCHAR FOR BIT DATA</td>
<td>— — — — — — — — — — — — — — — —</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLOB</td>
<td>— — — — — — — — — — — — — — —</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLOB</td>
<td>— — — — — — — — — — — — — — —</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DATE</td>
<td>— — — — — — Y Y — — — — — — Y —</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIME</td>
<td>— — — — — — Y Y — — — — — — Y —</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 5. Comparisons allowed by Cloudscape (continued). This table displays valid comparisons between data types in Cloudscape. A “Y” indicates that the comparison is allowed.

<table>
<thead>
<tr>
<th>Types</th>
<th>SMALL INT</th>
<th>INTEGER</th>
<th>DECIMAL</th>
<th>REAL</th>
<th>DOUBLE</th>
<th>FLOAT</th>
<th>CHAR</th>
<th>VARCHAR</th>
<th>LONG</th>
<th>VARCHA</th>
<th>FOR BIT</th>
<th>DATE</th>
<th>TIME</th>
<th>STAMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Y</td>
</tr>
<tr>
<td>STAMP</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Y</td>
</tr>
</tbody>
</table>
BIGINT

BIGINT provides 8 bytes of storage for integer values.

Syntax

BIGINT

Corresponding Compile-Time Java Type

`java.lang.Long`

JDBC Metadata Type (java.sql.Types)

BIGINT

Minimum Value

-922372036854775808 (`java.lang.Long.MIN_VALUE`)

Maximum Value

9223372036854775807 (`java.lang.Long.MAX_VALUE`)

When mixed with other data types in expressions, the resulting data type follows the rules shown in "Numeric Type Promotion in Expressions" on page 146.

An attempt to put an integer value of a larger storage size into a location of a smaller size fails if the value cannot be stored in the smaller-size location. Integer types can always successfully be placed in approximate numeric values, although with the possible loss of some precision. BIGINTs can be stored in DECIMALs if the DECIMAL precision is large enough for the value.

Examples

9223372036854775807
**BLOB**

A BLOB (binary large object) is a varying-length binary string that can be up to two gigabytes long. Like other binary types, BLOB strings are not associated with a code page. In addition, BLOB strings do not hold character data.

The length is given in bytes for BLOB unless one of the suffixes K, M, or G is given, relating to the multiples of 1024, 1024*1024, 1024*1024*1024 respectively.

**Note:** Length is specified in bytes for BLOB.

**Syntax**

```
{ BLOB | BINARY LARGE OBJECT } ( length [{K | M | G }])
```

**Corresponding Compile-Time Java Type**

`java.sql.Blob`

**JDBC Metadata Type (java.sql.Types)**

BLOB

Use the `getBlob` method on the `java.sql.ResultSet` to retrieve a BLOB handle to the underlying data.

**Related Information**

see "java.sql.Blob and java.sql.Clob" on page 215

**Examples**

```java
create table pictures(name varchar(32) not null primary key, pic blob(16M));
--find all logotype pictures
select length(pic), name from pictures where name like '%logo%';

--find all image doubles (blob comparisons)
select a.name as double_one, b.name as double_two
from pictures as a, pictures as b
where a.name < b.name
and a.pic = b.pic
order by 1,2;
```
CHAR

CHAR provides for fixed-length storage of strings.

Syntax

```
CHAR[ACTER] [(length)]
```

*length* is an unsigned integer constant. The default length for a CHAR is 1.

**Corresponding Compile-Time Java Type**

```
java.lang.String
```

**JDBC Metadata Type (java.sql.Types)**

CHAR

Cloudscape inserts spaces to pad a string value shorter than the expected length. Cloudscape truncates spaces from a string value longer than the expected length. Characters other than spaces cause an exception to be raised. When binary comparison operators are applied to CHARs, the shorter string is padded with spaces to the length of the longer string.

When CHARs and VARCHARS are mixed in expressions, the shorter value is padded with spaces to the length of the longer value.

The type of a string constant is CHAR.

**Implementation-Defined Aspects**

The only limit on the length of CHAR data types is the value `java.lang.Integer.MAX_VALUE`.

**Examples**

-- within a string constant use two single quotation marks
-- to represent a single quotation mark or apostrophe

VALUES 'hello this is Joe''s string'
CHAR FOR BIT DATA

A CHAR FOR BIT DATA type allows you to store byte strings of a specified length. It is useful for unstructured data where character strings are not appropriate.

Syntax

{ CHAR | CHARACTER }[(length)] FOR BIT DATA

length is an unsigned integer literal designating the length in bytes.

The default length for a CHAR FOR BIT DATA type is 1, and the maximum size of length is 254 bytes.

JDBC Metadata Type (java.sql.Types)

BINARY

CHAR FOR BIT DATA stores fixed-length byte strings. If a CHAR FOR BIT DATA value is smaller than the target CHAR FOR BIT DATA, it is padded with a 0x20 byte value.

Comparisons of CHAR FOR BIT DATA and VARCHAR FOR BIT DATA values are precise. For two bit strings to be equal, they must be exactly the same length. (This differs from the way some other DBMSs handle BINARY values but works as specified in SQL-92.)

An operation on a VARCHAR FOR BIT DATA and a CHAR FOR BIT DATA value (e.g., a concatenation) yields a VARCHAR FOR BIT DATA value.

Examples

CREATE TABLE t (b CHAR(2) FOR BIT DATA);
INSERT INTO t VALUES (X'DE');
SELECT *
FROM t;
-- yields the following output
B
-----
de20
CLOB

A CLOB (character large object) value can be up to two giga-characters long. A CLOB is used to store unicode character-based data, such as large documents in any character set.

The length is given in number characters for both CLOB, unless one of the suffixes K, M, or G is given, relating to the multiples of 1024, 1024*1024, 1024*1024*1024 respectively.

Length is specified in characters (unicode) for CLOB.

Syntax
{CLOB | CHARACTER LARGE OBJECT}(length [{K | M | G}])

Corresponding Compile-Time Java Type
java.sql.Clob

JDBC Metadata Type (java.sql.Types)
CLOB

Corresponding Compile-Time Java Type
java.sql.Clob

JDBC Metadata Type (java.sql.Types)
CLOB

Use the getClob method on the java.sql.ResultSet to retrieve a CLOB handle to the underlying data.

Related Information

Examples
import java.sql.*;

public class clob {
    public static void main(String[] args) {
        try {
            String url = "jdbc:derby:clobberyclob;create=true";
            Class.forName("org.apache.derby.jdbc.EmbeddedDriver").newInstance();
            Connection conn = DriverManager.getConnection(url);
            Statement s = conn.createStatement();
            s.executeUpdate("CREATE TABLE documents (id INT, text CLOB(64 K))");
            conn.commit();

            // --- add a file
            java.io.File file = new java.io.File("asciifile.txt");
            int fileLength = (int) file.length();

            // - first, create an input stream
            java.io.InputStream fin = new java.io.FileInputStream(file);
            PreparedStatement ps = conn.prepareStatement("INSERT INTO documents VALUES (?, ?)");
            ps.setInt(1, 1477);
// - set the value of the input parameter to the input stream
ps.setAsciiStream(2, fin, fileLength);
ps.execute();
conn.commit();

// --- reading the columns
ResultSet rs = s.executeQuery("SELECT text FROM documents 
WHERE id = 1477");
while (rs.next()) {
    java.sql.Clob aclob = rs.getClob(1);
    java.io.InputStream ip = rs.getAsciiStream(1);
    int c = ip.read();
    while (c > 0) {
        System.out.print((char)c);
        c = ip.read();
    }
    System.out.print("\n");
    // ...
}

} catch (Exception e) {
    System.out.println("Error! "+e);
}
}
DATE

DATE provides for storage of a year-month-day in the range supported by java.sql.Date.

Syntax
DATE

Corresponding Compile-Time Java Type
java.sql.Date

JDBC Metadata Type (java.sql.Types)
DATE

Dates, times, and timestamps must not be mixed with one another in expressions.

Any value that is recognized by the java.sql.Date method is permitted in a column of the corresponding SQL date/time data type.

Examples
VALUES DATE('1994-02-23')

VALUES '1993-09-01'
DECIMAL

DECIMAL provides an exact numeric in which the precision and scale can be arbitrarily sized. You can specify the precision (the total number of digits, both to the left and the right of the decimal point) and the scale (the number of digits of the fractional component). The amount of storage required is based on the precision.

Syntax

{ DECIMAL | DEC } [(precision [, scale ])]

The precision must be between 1 and 31. The scale must be less than or equal to the precision.

If the scale is not specified, the default scale is 0. If the precision is not specified, the default precision is 5.

An attempt to put a numeric value into a DECIMAL is allowed as long as any non-fractional precision is not lost. When truncating trailing digits from a DECIMAL value, Cloudscape rounds down.

For example:

-- this cast loses only fractional precision
values cast (1.798765 AS decimal(5,2));
1
---------
1.79

-- this cast does not fit
values cast (1798765 AS decimal(5,2));
1
---------
ERROR 22003: The resulting value is outside the range for the data type DECIMAL/NUMERIC(5,2).

When mixed with other data types in expressions, the resulting data type follows the rules shown in "Numeric Type Promotion in Expressions" on page 146.

See also "Storing Values of One Numeric Data Type in Columns of Another Numeric Data Type" on page 147.

When two decimal values are mixed in an expression, the scale and precision of the resulting value follow the rules shown in "Scale for Decimal Arithmetic" on page 147.

Corresponding Compile-Time Java Type

java.math.BigDecimal

JDBC Metadata Type (java.sql.Types)

DECIMAL

Examples

VALUES 123.456

VALUES 0.001

Integer constants too big for BIGINT are made DECIMAL constants.
DOUBLE

The DOUBLE data type is a synonym for the "DOUBLE PRECISION" on page 162 data type.

Syntax
DOUBLE
DOUBLE PRECISION

The DOUBLE PRECISION data type provides 8-byte storage for numbers using IEEE floating-point notation.

Syntax

DOUBLE PRECISION

or, alternately

DOUBLE

DOUBLE can be used synonymously with DOUBLE PRECISION.

Limitations

DOUBLE value ranges:

- Smallest DOUBLE value: -1.79769E+308
- Largest DOUBLE value: 1.79769E+308
- Smallest positive DOUBLE value: 2.225E-307
- Largest negative DOUBLE value: -2.225E-307

These limits are different from the java.lang.Double Java type limits.

An exception is thrown when any double value is calculated or entered that is outside of these value ranges. Arithmetic operations do not round their resulting values to zero. If the values are too small, you will receive an exception.

Numeric floating point constants are limited to a length of 30 characters.

-- this example will fail because the constant is too long: 01234567890123456789012345678901E0;

Corresponding Compile-Time Java Type

java.lang.Double

JDBC Metadata Type (java.sql.Types)

DOUBLE

When mixed with other data types in expressions, the resulting data type follows the rules shown in "Numeric Type Promotion in Expressions" on page 146.

See also "Storing Values of One Numeric Data Type in Columns of Another Numeric Data Type" on page 147.

Examples

3421E+09
425.43E9
9E-10
4356267544.32333E+30
FLOAT

The FLOAT data type is an alias for a REAL or DOUBLE PRECISION data type, depending on the precision you specify.

Syntax

FLOAT [ (precision) ]

The default precision for FLOAT is 53 and is equivalent to DOUBLE PRECISION. A precision of 23 or less makes FLOAT equivalent to REAL. A precision of 24 or greater makes FLOAT equivalent to DOUBLE PRECISION. If you specify a precision of 0, you get an error. If you specify a negative precision, you get a syntax error.

JDBC Metadata Type (java.sql.Types)

REAL or DOUBLE

Limitations

If you are using a precision of 24 or greater, the limits of FLOAT are similar to the limits of DOUBLE.

If you are using a precision of 23 or less, the limits of FLOAT are similar to the limits of REAL.
INTEGER

INTEGER provides 4 bytes of storage for integer values.

Syntax
{ INTEGER | INT }

Corresponding Compile-Time Java Type
java.lang.Integer

JDBC Metadata Type (java.sql.Types)
INTEGER

Minimum Value
-2147483648 (java.lang.Integer.MIN_VALUE)

Maximum Value
2147483647 (java.lang.Integer.MAX_VALUE)

When mixed with other data types in expressions, the resulting data type follows the rules shown in "Numeric Type Promotion in Expressions" on page 146.

See also "Storing Values of One Numeric Data Type in Columns of Another Numeric Data Type" on page 147.

Examples
3453
425
LONG VARCHAR

The LONG VARCHAR type allows storage of character strings of unlimited length. It is identical to VARCHAR, except that you do not have to specify a maximum length when creating columns of this type.

Syntax

LONG VARCHAR

Corresponding Compile-Time Java Type

java.lang.String

JDBC Metadata Type (java.sql.Types)

LONGVARCHAR

When you are converting from Java values to SQL values, no Java type corresponds to LONG VARCHAR.
LONG VARCHAR FOR BIT DATA

The LONG VARCHAR FOR BIT DATA type allows storage of bit strings up to 32,700 bytes. It is identical to "VARCHAR FOR BIT DATA" on page 173 except that you do not have to specify a maximum length when creating columns of this type.

Syntax

LONG VARCHAR FOR BIT DATA
NUMERIC

NUMERIC is a synonym for DECIMAL and behaves the same way. See "DECIMAL" on page 160.

Syntax
NUMERIC [(precision [, scale ])]

Corresponding Compile-Time Java Type
java.math.BigDecimal

JDBC Metadata Type (java.sql.Types)
NUMERIC

Examples
123.456
.001
REAL

The REAL data type provides 4 bytes of storage for numbers using IEEE floating-point notation.

Syntax
REAL

Corresponding Compile-Time Java Type
java.lang.Float

JDBC Metadata Type (java.sql.Types)
REAL

Limitations
REAL value ranges:
- Smallest REAL value: -3.402E+38
- Largest REAL value: 3.402E+38
- Smallest positive REAL value: 1.175E-37
- Largest negative REAL value: -1.175E-37

These limits are different from the java.lang.Float Java type limits.

An exception is thrown when any double value is calculated or entered that is outside of these value ranges. Arithmetic operations do not round their resulting values to zero. If the values are too small, you will receive an exception. The arithmetic operations take place with double arithmetic in order to detect underflows.

Numeric floating point constants are limited to a length of 30 characters.
-- this example will fail because the constant is too long:
values 01234567890123456789012345678901e0;

When mixed with other data types in expressions, the resulting data type follows the rules shown in "Numeric Type Promotion in Expressions" on page 146.

See also "Storing Values of One Numeric Data Type in Columns of Another Numeric Data Type" on page 147.

Constants always map to DOUBLE PRECISION; use a CAST to convert a constant to a REAL.
SMALLINT

SMALLINT provides 2 bytes of storage.

Syntax
SMALLINT

Corresponding Compile-Time Java Type
java.lang.Short

JDBC Metadata Type (java.sql.Types)
SMALLINT

Minimum Value
-32768 (java.lang.Short.MIN_VALUE)

Maximum Value
32767 (java.lang.Short.MAX_VALUE)

When mixed with other data types in expressions, the resulting data type follows the rules shown in "Numeric Type Promotion in Expressions" on page 146.

See also "Storing Values of One Numeric Data Type in Columns of Another Numeric Data Type" on page 147.

Constants in the appropriate format always map to INTEGER or BIGINT, depending on their length.
**TIME**

TIME provides for storage of an hour-minutes-seconds-fractional-seconds value with six digits in the fractional-seconds part. Cloudscape also supports the ISO format of 8 characters (6 digits, and 2 decimal points).

**Syntax**

```java
TIME
```

**Corresponding Compile-Time Java Type**

```java
java.sql.Time
```

**JDBC Metadata Type (java.sql.Types)**

TIME

Dates, times, and timestamps cannot be mixed with one another in expressions except with a CAST.

Any value that is recognized by the `java.sql.Time` method is permitted in a column of the corresponding SQL date/time data type.

**Related Information**

```sql
TIME 'HH:MM:SS[.FFFFFF']
```

**Examples**

```sql
VALUES TIME('15:09:02')
VALUES '15:09:02'
```
TIMESTAMP

TIMESTAMP stores a combined DATE and TIME value to be stored. It permits a fractional-seconds value of up to nine digits. Cloudscape also supports the ISO format of 23 characters (17 digits, 3 dashes, and 3 decimal points).

Syntax

```
TIMESTAMP
```

Corresponding Compile-Time Java Type

```
java.sql.Timestamp
```

JDBC Metadata Type (java.sql.Types)

```
TIMESTAMP
```

Dates, times, and timestamps cannot be mixed with one another in expressions.

Timestamp only accepts formats accepted by Timestamp.valueOf. Any value that is recognized by the `java.sql.Timestamp.valueOf(String)` method is permitted in a column of the corresponding SQL date/time data type.

Examples

```
VALUES '1960-01-01 23:03:20'
VALUES TIMESTAMP('1962-09-23 03:23:34.234')
VALUES TIMESTAMP('1960-01-01 23:03:20')
```
**VARCHAR**

VARCHAR provides for variable-length storage of strings.

**Syntax**

```java
{ VARCHAR | CHAR VARYING | CHARACTER VARYING }(length)
```

`length` is an unsigned integer constant, and it must not be greater than the constraint of the integer used to specify the length, the value `java.lang.Integer.MAX_VALUE`.

**Corresponding Compile-Time Java Type**

`java.lang.String`

**JDBC Metadata Type (java.sql.Types)**

VARCHAR

Cloudscape does not pad a VARCHAR value whose length is less than specified. Cloudscape truncates spaces from a string value when a length greater than the VARCHAR expected is provided. Characters other than spaces are not truncated, and instead cause an exception to be raised. When binary comparison operators are applied to VARCHARs, the lengths of the operands are not altered, and spaces at the end of the values are ignored.

When CHARs and VARCHARs are mixed in expressions, the shorter value is padded with spaces to the length of the longer value.

The type of a string constant is CHAR, not VARCHAR.
VARCHARM FOR BIT DATA

The VARCHARM FOR BIT DATA type allows you to store binary strings less than or equal to a specified length. It is useful for unstructured data where character strings are not appropriate (e.g., images).

Syntax

{ VARCHAR | CHAR VARYING | CHARACTER VARYING } (length) FOR BIT DATA

length is an unsigned integer literal designating the length in bytes.

Unlike the case for the CHAR FOR BIT DATA type, there is no default length for a VARCHARM FOR BIT DATA type. The maximum size of the length value is 32,672 bytes.

JDBC Metadata Type (java.sql.Types)

VARBINARY

VARCHARM FOR BIT DATA stores variable-length byte strings. Unlike CHAR FOR BIT DATA values, VARCHARM FOR BIT DATA values are not padded out to the target length.

An operation on a VARCHARM FOR BIT DATA and a CHAR FOR BIT DATA value (e.g., a concatenation) yields a VARCHARM FOR BIT DATA value.

Examples

The type of a byte literal is always a VARCHARM FOR BIT DATA, not a CHAR FOR BIT DATA.
SQL Expressions

Syntax for many statements and expressions includes the term Expression, or a term for a specific kind of expression such as TableSubquery. Expressions are allowed in these specified places within statements. Some locations allow only a specific type of expression or one with a specific property. Table 6 on page 174 lists all the possible SQL expressions and indicates where they are allowed.

If not otherwise specified, an expression is permitted anywhere the word Expression appears in the syntax. This includes:

- SelectExpression
- UPDATE statement (SET portion)
- VALUES expression
- WHERE clause

Of course, many other statements include these elements as building blocks, and so allow expressions as part of these elements.

Table 6. Table of Expressions

<table>
<thead>
<tr>
<th>Expression Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>General expressions</td>
<td>All expressions that might result in a value of any type.</td>
</tr>
<tr>
<td>Column reference</td>
<td>A [column–Name] that references the value of the column made visible to the expression containing the Column reference. You must qualify the column–Name by the table name or correlation name if it is ambiguous. The qualifier of a [column–Name] must be the correlation name, if a correlation name is given to a table that is in a FROM clause. The table name is no longer visible as a column–Name qualifier once it has been aliased by a correlation name.</td>
</tr>
<tr>
<td>Constant</td>
<td>Most built-in data types typically have constants associated with them (as shown in “Data Types” on page 145).</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL is an untyped constant representing the unknown value.</td>
</tr>
</tbody>
</table>
### Table 6. Table of Expressions (continued)

<table>
<thead>
<tr>
<th>Expression Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic parameter</td>
<td>A dynamic parameter is a parameter to an SQL statement for which the value is not specified when the statement is created. Instead, the statement has a question mark (?) as a placeholder for each dynamic parameter. See &quot;Dynamic Parameters&quot; on page 179. Dynamic parameters are permitted only in prepared statements. You must specify values for them before the prepared statement is executed. The values specified must match the types expected.</td>
</tr>
<tr>
<td>CAST expression</td>
<td>Lets you specify the type of NULL or of a dynamic parameter or convert a value to another type. See &quot;CAST&quot; on page 89.</td>
</tr>
<tr>
<td>scalar subquery</td>
<td>Subquery that returns a single row with a single column. See ScalarSubquery.</td>
</tr>
<tr>
<td>table subquery</td>
<td>Subquery that returns more than one column and more than one row. See TableSubquery.</td>
</tr>
<tr>
<td>Conditional expression</td>
<td>A conditional expression chooses an expression to evaluate based on a boolean test.</td>
</tr>
<tr>
<td><strong>Boolean expressions</strong></td>
<td>Expressions that result in boolean values. Most general expressions can result in boolean values.</td>
</tr>
<tr>
<td></td>
<td>Boolean expressions commonly used in a WHERE clause are made of operands operated on by SQL operators. See Table 7 on page 177.</td>
</tr>
<tr>
<td><strong>Numeric expressions</strong></td>
<td>Expressions that result in numeric values. Most of the general expressions can result in numeric values.</td>
</tr>
<tr>
<td></td>
<td>Numeric values have one of the following types:</td>
</tr>
<tr>
<td></td>
<td>SMALLINT, INTEGER, BIGINT, REAL, DOUBLE PRECISION, DECIMAL.</td>
</tr>
<tr>
<td>+, -, *, /, unary + and - expressions</td>
<td>+, -, *, /, unary + and -</td>
</tr>
<tr>
<td></td>
<td>Evaluate the expected math operation on the operands. If both operands are the same type, the result type is not promoted, so the division operator on integers results in an integer that is the truncation of the actual numeric result. When types are mixed, they are promoted as described in &quot;Data Types&quot; on page 145. Unary + is a noop (i.e., +4 is the same as 4). Unary - is the same as multiplying the value by -1, effectively changing its sign.</td>
</tr>
</tbody>
</table>
### Expression Precedence

Precedence of operations from highest to lowest is:

- (), ?, Constant (including sign), NULL, ColumnReference, ScalarSubquery, CAST
- LENGTH, CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, and other built-ins

---

<table>
<thead>
<tr>
<th>Expression Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG</td>
<td>Returns the average of a set of numeric values. [“AVG” on page 87]</td>
</tr>
<tr>
<td>SUM</td>
<td>Returns the sum of a set of numeric values. [“SUM” on page 128]</td>
</tr>
<tr>
<td>LENGTH</td>
<td>Returns the number of characters in a character or bit string. See [“LENGTH” on page 93]</td>
</tr>
<tr>
<td>LOWER</td>
<td>See [“LCASE or LOWER” on page 115]</td>
</tr>
<tr>
<td>COUNT</td>
<td>Returns the count of a set of values. See [“COUNT” on page 96, “COUNT(*)” on page 97]</td>
</tr>
</tbody>
</table>

Character expressions

- A CHAR or VARCHAR value that uses wildcards. The wildcards % and _ make a character string a pattern against which the LIKE operator can look for a match. See [“LIKE” on page 178] |

Concatenation expression

- In a concatenation expression, the concatenation operator, “||”, concatenates its right operand to the end of its left operand. Operates on character and bit strings. See [“Concatenation” on page 94] |

Built-in string functions

- The built-in string functions act on a String and return a string. See [“LTRIM” on page 116, “LCASE or LOWER” on page 115, “RTRIM” on page 122, “SUBSTR” on page 127, and “UCASE or UPPER” on page 131] |

User functions

- User functions return information about the current user as a String. See [“CURRENT_USER” on page 106, “SESSION_USER” on page 124, and “USER” on page 132] |

Date/time expressions

- A date/time expression results in a DATE, TIME, or TIMESTAMP value. Most of the general expressions can result in a date/time value. |

- CURRENT_DATE
  - Returns the current date. See [“CURRENT_DATE” on page 99] |

- CURRENT_TIME
  - Returns the current time. See [“CURRENT_TIME” on page 103] |

- CURRENT_TIMESTAMP
  - Returns the current timestamp. See [“CURRENT_TIMESTAMP” on page 105] |
• unary + and -
• *, /, || (concatenation)
• binary + and -
• comparisons, Quantified comparisons, EXISTS, IN, IS NULL, LIKE, BETWEEN, IS
• NOT
• AND
• OR

You can explicitly specify precedence by placing expressions within parentheses. An expression within parentheses is evaluated before any operations outside the parentheses are applied to it.

**Example**

\[(3+4)*9\]  
\[(age < 16 \text{ OR } age > 65) \text{ AND } employed = \text{TRUE}\]

**Boolean expression**

Boolean expressions are allowed in WHERE clauses and in check constraints. Boolean expressions in check constraints have limitations not noted here; see "CONSTRAINT clause" on page 43 for more information. Boolean expressions in a WHERE clause have a highly liberal syntax; see "WHERE clause" on page 81, for example.

A boolean expression can include a boolean operator or operators. These are listed in Table 7 on page 177.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Explanation and Example</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND, OR, NOT</td>
<td>Evaluate any operand(s) that are boolean expressions</td>
<td>{ Expression AND</td>
</tr>
<tr>
<td></td>
<td>(orig_airport = 'SFO') OR (dest_airport = 'GRU')</td>
<td>Expression OR</td>
</tr>
<tr>
<td></td>
<td>-- returns true</td>
<td>NOT Expression</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operator</th>
<th>Explanation and Example</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparisons</td>
<td>&lt;, =, &gt;, &lt;=, =&gt;, &lt;&gt; are applicable to all of the built-in types.</td>
<td>{ Expression</td>
</tr>
<tr>
<td></td>
<td>DATE'1998-02-26' &lt; DATE'1998-03-01'</td>
<td>&lt;=</td>
</tr>
<tr>
<td></td>
<td>-- returns true</td>
<td>&gt;=</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operator</th>
<th>Explanation and Example</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS NULL, IS NOT NULL</td>
<td>Test whether the result of an expression is null or not.</td>
<td>Expression IS [ NOT ] NULL</td>
</tr>
</tbody>
</table>

**Table 7. SQL Boolean Operators**
### Table 7. SQL Boolean Operators (continued)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Explanation and Example</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIKE</td>
<td>Attempts to match a character expression to a character pattern, which is a character string that includes one or more wildcards. % matches any number (zero or more) of characters in the corresponding position in first character expression. _ matches one character in the corresponding position in the character expression. Any other character matches only that character in the corresponding position in the character expression. city LIKE 'Sant_' To treat % or _ as constant characters, escape the character with an optional escape character, which you specify with the ESCAPE clause. SELECT a FROM tabA WHERE a LIKE '%=_' ESCAPE '='</td>
<td>CharacterExpression [ NOT ] LIKE CharacterExpression WithWildCard [ ESCAPE 'escapeCharacter' ]</td>
</tr>
<tr>
<td>BETWEEN</td>
<td>Tests whether the first operand is between the second and third operands. The second operand must be less than the third operand. Applicable only to types to which &lt;= and &gt;= can be applied. WHERE booking_date BETWEEN DATE'1998-02-26' AND DATE'1998-03-01</td>
<td>Expression [ NOT ] BETWEEN Expression AND Expression</td>
</tr>
<tr>
<td>IN</td>
<td>Operates on table subquery or list of values. Returns TRUE if the left expression's value is in the result of the table subquery or in the list of values. Table subquery can return multiple rows but must return a single column. WHERE booking_date NOT IN (SELECT booking_date FROM HotelBookings WHERE rooms_available = 0)</td>
<td>{ Expression [ NOT ] IN TableSubquery Expression [ NOT ] IN ( Expression [, Expression ]* ) }</td>
</tr>
<tr>
<td>EXISTS</td>
<td>Operates on a table subquery. Returns TRUE if the table subquery returns any rows, and FALSE if it returns no rows. Table subquery can return multiple columns (only if you use * to denote multiple columns) and rows. WHERE EXISTS (SELECT * FROM Flights WHERE dest_airport = 'SFO' AND orig_airport = 'GRU')</td>
<td>[NOT] EXISTS TableSubquery</td>
</tr>
</tbody>
</table>
Table 7. SQL Boolean Operators (continued)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Explanation and Example</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quantified comparison</td>
<td>A quantified comparison is a comparison operator (&lt;, =, &gt;, &lt;=, &gt;=, &lt;&gt;) with ALL or ANY or SOME applied. Operates on table subqueries, which can return multiple rows but must return a single column. If ALL is used, the comparison must be true for all values returned by the table subquery. If ANY or SOME is used, the comparison must be true for at least one value of the table subquery. ANY and SOME are equivalent. WHERE normal_rate &lt; ALL (SELECT budget/550 FROM Groups)</td>
<td></td>
</tr>
</tbody>
</table>

Dynamic Parameters

You can prepare statements that are allowed to have parameters for which the value is not specified when the statement is prepared using PreparedStatement methods in the JDBC API. These parameters are called dynamic parameters and are represented by a ?.

The JDBC API documents refer to dynamic parameters as IN, INOUT, or OUT parameters. In SQL, they are always IN parameters.

New: Cloudscape 10 supports the interface ParameterMetaData, new in JDBC 3.0. This interface describes the number, type, and properties of prepared statement parameters. See the IBM Cloudscape Developer’s Guide for more information.

You must specify values for them before executing the statement. The values specified must match the types expected.

Dynamic Parameters Example

```java
PreparedStatement ps2 = conn.prepareStatement("UPDATE HotelAvailability SET rooms_available = " +
    
    (rooms_available - ?) WHERE hotel_id = ? " +
    "AND booking_date BETWEEN ? AND ?")
-- this sample code sets the values of dynamic parameters
-- to the values of program variables
ps2.setInt(1, numberRooms);
ps2.setInt(2, theHotel.hotelId);
ps2.setDate(3, arrival);
ps2.setDate(4, departure);
updateCount = ps2.executeUpdate();
```

Where Dynamic Parameters Are Allowed

You can use dynamic parameters anywhere in an expression where their data type can be easily deduced.

1. Use as the first operand of BETWEEN is allowed if one of the second and third operands is not also a dynamic parameter. The type of the first operand is assumed to be the type of the non-dynamic parameter, or the union result of their types if both are not dynamic parameters.

WHERE ? BETWEEN DATE('1996-01-01') AND ?
-- types assumed to be DATES
2. Use as the second or third operand of BETWEEN is allowed. Type is assumed to be the type of the left operand.

   WHERE DATE('1996-01-01') BETWEEN ? AND ?
   -- types assumed to be DATES

3. Use as the left operand of an IN list is allowed if at least one item in the list is not itself a dynamic parameter. Type for the left operand is assumed to be the union result of the types of the non-dynamic parameters in the list.

   WHERE ? NOT IN (?, ?, 'Santiago')
   -- types assumed to be CHAR

4. Use in the values list in an IN predicate is allowed if the first operand is not a dynamic parameter or its type was determined in rule 3. Type of the dynamic parameters appearing in the values list is assumed to be the type of the left operand.

   WHERE FloatColumn IN (?, ?, ?)
   -- types assumed to be FLOAT

5. For the binary operators +, -, *, /, AND, OR, <, >, =, <>, <=, and >=, use of a dynamic parameter as one operand but not both is permitted. Its type is taken from the other side.

   WHERE ? < CURRENT_TIMESTAMP
   -- type assumed to be a TIMESTAMP

6. Use in a CAST is always permitted. This gives the dynamic parameter a type.

   CALL valueOf(CAST (?) AS VARCHAR(10)))

7. Use on either or both sides of LIKE operator is permitted. When used on the left, the type of the dynamic parameter is set to the type of the right operand, but with the maximum allowed length for the type. When used on the right, the type is assumed to be of the same length and type as the left operand.

   (LIKE is permitted on CHAR and VARCHAR types; see "Concatenation" on page 94 and "LIKE" on page 178 for more information.)

   WHERE ? LIKE 'Santi%'
   -- type assumed to be CHAR with a length of
   -- java.lang.Integer.MAX_VALUE

8. A ? parameter is allowed by itself on only one side of the || operator. That is, "? || ?" is not allowed. The type of a ? parameter on one side of a || operator is determined by the type of the expression on the other side of the || operator. If the expression on the other side is a CHAR or VARCHAR, the type of the parameter is VARCHAR with the maximum allowed length for the type. If the expression on the other side is a CHAR FOR BIT DATA or VARCHAR FOR BIT DATA type, the type of the parameter is VARCHAR FOR BIT DATA with the maximum allowed length for the type.

   SELECT BITColumn || ?
   FROM UserTable
   -- Type assumed to be CHAR FOR BIT DATA of length specified for BITColumn

9. In a conditional expression, which uses a ?, use of a dynamic parameter (which is also represented as a ?) is allowed. The type of a dynamic parameter as the first operand is assumed to be boolean. Only one of the second and third operands can be a dynamic parameter, and its type will be assumed to be the same as that of the other (that is, the third and second operand, respectively).

   SELECT c1 IS NULL ? ? : c1
   -- allows you to specify a "default" value at execution time
   -- dynamic parameter assumed to be the type of c1
   -- you cannot have dynamic parameters on both sides
   -- of the :

10. A dynamic parameter is allowed as an item in the values list or select list of an INSERT statement. The type of the dynamic parameter is assumed to be
the type of the target column. A ? parameter is not allowed by itself in any select list, including the select list of a subquery, unless there is a corresponding column in a UNION (see no. [16] below) that is not dynamic.

```
INSERT INTO t VALUES (?)
-- dynamic parameter assumed to be the type
-- of the only column in table t
INSERT INTO t SELECT ?
FROM t2
-- not allowed
```

11. A ? parameter in a comparison with a subquery takes its type from the expression being selected by the subquery. For example:

```
SELECT *
FROM tab1
WHERE ? = (SELECT x FROM tab2)
```

```
SELECT *
FROM tab1
WHERE ? = ANY (SELECT x FROM tab2)
-- In both cases, the type of the dynamic parameter is 
-- assumed to be the same as the type of tab2.x.
```

12. A dynamic parameter is allowed as the value in an UPDATE statement. The type of the dynamic parameter is assumed to be the type of the column in the target table.

```
UPDATE t2 SET c2 =? -- type is assumed to be type of c2
```

13. A dynamic parameter is not allowed as the operand of the unary operators - or +.

14. LENGTH allow a dynamic parameter. The type is assumed to be a maximum length VARCHAR type.

```
SELECT LENGTH(?)
```

15. Quantified comparison.

```
? = SOME (SELECT 1 FROM t)
-- is valid. Dynamic parameter assumed to be INTEGER type
I = SOME (SELECT ? FROM t)
-- is valid. Dynamic parameter assumed to be INTEGER type.
```

16. A dynamic parameter is allowed to represent a column if it appears in a UNION expression; Cloudscape can infer the data type from the corresponding column in the UNION.

```
SELECT ?
FROM t
UNION SELECT 1
FROM t
-- dynamic parameter assumed to be INT
VALUES 1 UNION VALUES ?
-- dynamic parameter assumed to be INT
```

17. A dynamic parameter is allowed as the left operand of an IS expression and is assumed to be a boolean.

Once the type of a dynamic parameter is determined based on the expression it is in, that expression is allowed anywhere it would normally be allowed if it did not include a dynamic parameter. For example, above we said that a dynamic parameter cannot be used as the operand of a unary -. It can, however, appear within an expression that is the operand of a unary minus, such as:

```
- (1+?)
```

The dynamic parameter is assumed to be an INTEGER (because the binary operator +’s other operand is of the type INT). Because we know its type, it is allowed as the operand of a unary -.
Chapter 2. SQL Reserved Words

This section lists all the Cloudscape reserved words, including those in the SQL-92 standard. Cloudscape will return an error if you use any of these keywords as an identifier name unless you surround the identifier name with quotes (''). See "Rules for SQL92Identifiers" on page 2.

ADD
ALL
ALLOCATE
ALTER
AND
ANY
ARE
AS
ASC
ASSERTION
AT
AUTHORIZATION
AVG
BEGIN
BETWEEN
BIT
BIT_LENGTH
BOOLEAN
BOTH
BY
CALL
CASCADE
CASCADED
CASE
CAST
CHAR
CHARACTER
CHARACTER_LENGTH
CHAR_LENGTH
CHECK
CLOSE
COLLATE
COLLATION
COLUMN
COMMIT
CONNECT
CONNECTION
CONSTRAINT
CONSTRAINTS
CONTINUE
CONVERT
CORRESPONDING
COUNT
CREATE
CROSS
CURRENT
CURRENT_DATE
INPUT
INSENSITIVE
INSERT
INT
INTEGER
INTERSECT
INTO
IS
ISOLATION
JOIN
KEY
LAST
LEADING
LEFT
LIKE
LOCAL
LONGINT
LOWER
LTRIM
MATCH
MAX
MIN
MINUTE
NATIONAL
NATURAL
NCHAR
NVARCHAR
NEXT
NO
NOT
NULL
NULLIF
NUMERIC
OCTET_LENGTH
OF
ON
ONLY
OPEN
OPTION
OR
ORDER
OUT
OUTER
OUTPUT
OVERLAPS
PAD
PARTIAL
PREPARE
PRESERVE
PRIMARY
PRIOR
PRIVILEGES
PROCEDURE
PUBLIC
READ
REAL
REFERENCES
RELATIVE
RESTRICT
REVOKE
RIGHT
ROLLBACK
ROWS
RTRIM
RUNTIMESTATISTICS
SCHEMA
SCROLL
SECOND
SELECT
SESSION_USER
SET
SMALLINT
SOME
SPACE
SQL
SQLCODE
SQLERROR
SQLSTATE
SUBSTR
SUBSTRING
SUM
SYSTEM_USER
TABLE
TEMPORARY
TIMEZONE_HOUR
TIMEZONE_MINUTE
TINYINT
TO
TRAILING
TRANSACTION
TRANSLATE
TRANSLATION
TRIM
TRUE
UNION
UNIQUE
UNKNOWN
UPDATE
UPPER
USER
USING
VALUES
VARCHAR
VARYING
VIEW
WHENEVER
WHERE
WITH
WORK
WRITE
YEAR
Chapter 3. Cloudscape Support for SQL-92 Features

Table 8 shows the SQL-92 features that Cloudscape 10 supports. There are four levels of SQL-92 support:

- **SQL92E**
  Entry
- **SQL92T**
  Transitional, a level defined by NIST in a publication called FIPS 127-2
- **SQL92I**
  Intermediate
- **SQL92F**
  Full

<table>
<thead>
<tr>
<th>Feature</th>
<th>Source</th>
<th>Cloudscape 10</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic types</strong></td>
<td>SQL92E</td>
<td>yes</td>
</tr>
<tr>
<td>SMALLINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DECIMAL(p,s)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NUMERIC(p,s)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLOAT(p)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHAR(n)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Basic math operations</strong></td>
<td>SQL92E</td>
<td>yes</td>
</tr>
<tr>
<td>+, *, -, /, unary +, unary -</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Basic comparisons</strong></td>
<td>SQL92E</td>
<td>yes</td>
</tr>
<tr>
<td>&lt;, &gt;, &lt;=, &gt;=, &lt;&gt;, =</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Basic predicates</strong></td>
<td>SQL92E</td>
<td></td>
</tr>
<tr>
<td>BETWEEN, LIKE, NULL</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Quantified predicates</strong></td>
<td>SQL92E</td>
<td></td>
</tr>
<tr>
<td>IN, ALL/SOME, EXISTS</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td><strong>schema definition</strong></td>
<td>SQL92E</td>
<td></td>
</tr>
<tr>
<td>tables</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>views</td>
<td></td>
<td>yes (not updatable)</td>
</tr>
<tr>
<td>privileges</td>
<td></td>
<td>no, but you can configure databases for user authentication and restriction to read-only access</td>
</tr>
<tr>
<td><strong>column attributes</strong></td>
<td>SQL92E</td>
<td></td>
</tr>
<tr>
<td>default values</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>nullability</td>
<td></td>
<td>yes</td>
</tr>
</tbody>
</table>
Table 8. Support for SQL-92 Features (continued)

<table>
<thead>
<tr>
<th>Feature</th>
<th>Source</th>
<th>Cloudscape 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraints (non-deferrable)</td>
<td>SQL92E</td>
<td>yes (not stored in SYSCONSTRAINTS)</td>
</tr>
<tr>
<td>NOT NULL</td>
<td>SQL92E</td>
<td>yes</td>
</tr>
<tr>
<td>UNIQUE/PRIMARY KEY</td>
<td>SQL92E</td>
<td>yes</td>
</tr>
<tr>
<td>FOREIGN KEY</td>
<td>SQL92E</td>
<td>yes</td>
</tr>
<tr>
<td>CHECK</td>
<td>SQL92E</td>
<td>yes</td>
</tr>
<tr>
<td>View WITH CHECK OPTION</td>
<td>SQL92E</td>
<td>no, since views are not updatable</td>
</tr>
<tr>
<td>Delimited identifiers</td>
<td>SQL92E</td>
<td>yes</td>
</tr>
<tr>
<td>Correlated subqueries</td>
<td>SQL92E</td>
<td>yes</td>
</tr>
<tr>
<td>Cursors</td>
<td>SQL92E</td>
<td>done through JDBC</td>
</tr>
<tr>
<td>DECLARE, OPEN, FETCH, CLOSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UPDATE, DELETE CURRENT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insert, Update, Delete statements</td>
<td>SQL92E</td>
<td>yes</td>
</tr>
<tr>
<td>Joins</td>
<td>SQL92E</td>
<td>yes</td>
</tr>
<tr>
<td>Where qualifications</td>
<td>SQL92E</td>
<td>yes</td>
</tr>
<tr>
<td>Group by</td>
<td>SQL92E</td>
<td>yes</td>
</tr>
<tr>
<td>Having</td>
<td>SQL92E</td>
<td>yes</td>
</tr>
<tr>
<td>Aggregate functions</td>
<td>SQL92E</td>
<td>yes</td>
</tr>
<tr>
<td>Order by</td>
<td>SQL92E</td>
<td>yes</td>
</tr>
<tr>
<td>Select expressions</td>
<td>SQL92E</td>
<td>yes</td>
</tr>
<tr>
<td>Select *</td>
<td>SQL92E</td>
<td>yes</td>
</tr>
<tr>
<td>SQLCODE</td>
<td>SQL92E</td>
<td>no, deprecated in SQL-92</td>
</tr>
<tr>
<td>SQLSTATE</td>
<td>SQL92E</td>
<td>yes</td>
</tr>
<tr>
<td>Dynamic SQL 1</td>
<td>SQL92T</td>
<td>done through JDBC</td>
</tr>
<tr>
<td>ALLOCATE/DEALLOCATE/GET/SET DESCRIPTOR</td>
<td>SQL92T</td>
<td>done through JDBC</td>
</tr>
<tr>
<td>PREPARE/EXECUTE/EXECUTE IMMEDIATE</td>
<td>SQL92T</td>
<td>done through JDBC</td>
</tr>
<tr>
<td>DECLARE, OPEN, FETCH, CLOSE, UPDATE, DELETE dynamic cursor</td>
<td>SQL92T</td>
<td>done through JDBC</td>
</tr>
<tr>
<td>DESCRIBE output</td>
<td>SQL92T</td>
<td></td>
</tr>
<tr>
<td>Basic information schema</td>
<td>SQL92T</td>
<td>SYS.SYSTABLES, SYS.SYSVIEWS, SYS.SYSCOLUMNS</td>
</tr>
<tr>
<td>TABLES</td>
<td>SQL92T</td>
<td></td>
</tr>
<tr>
<td>VIEWS</td>
<td>SQL92T</td>
<td></td>
</tr>
<tr>
<td>COLUMNS</td>
<td>SQL92T</td>
<td></td>
</tr>
</tbody>
</table>
### Table 8. Support for SQL-92 Features (continued)

<table>
<thead>
<tr>
<th>Feature</th>
<th>Source</th>
<th>Cloudscape 10</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic schema manipulation</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CREATE/DROP TABLE</td>
<td>SQL92T</td>
<td>yes</td>
</tr>
<tr>
<td>CREATE/DROP VIEW</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>GRANT/REVOKE</td>
<td></td>
<td>no</td>
</tr>
<tr>
<td>ALTER TABLE ADD COLUMN</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>ALTER TABLE DROP COLUMN</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td><strong>Joined table</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INNER JOIN</td>
<td>SQL92T</td>
<td>yes</td>
</tr>
<tr>
<td>natural join</td>
<td></td>
<td>no</td>
</tr>
<tr>
<td>LEFT, RIGHT OUTER JOIN</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>join condition</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>named columns join</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td><strong>DATETIME data types 1</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>simple DATE, TIME, TIMESTAMP, INTERVAL</td>
<td>SQL92T</td>
<td>yes, not INTERVAL</td>
</tr>
<tr>
<td>datetime constants</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>datetime math</td>
<td></td>
<td>can do with Java methods</td>
</tr>
<tr>
<td>datetime comparisons</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>predicates: OVERLAPS</td>
<td></td>
<td>can do with Java methods</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>SQL92T</td>
<td>yes</td>
</tr>
<tr>
<td>LENGTH</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>concatenation (</td>
<td></td>
<td>)</td>
</tr>
<tr>
<td>UNION in views</td>
<td>SQL92T</td>
<td>yes</td>
</tr>
<tr>
<td>Implicit numeric casting</td>
<td>SQL92T</td>
<td>yes</td>
</tr>
<tr>
<td>Implicit character casting</td>
<td>SQL92T</td>
<td>yes</td>
</tr>
<tr>
<td><strong>Transaction isolation</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>READ WRITE/READ ONLY</td>
<td>SQL92T</td>
<td>through JDBC, database properties, and storage media.</td>
</tr>
<tr>
<td>RU, RC, RR, SER</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>Get diagnostics</td>
<td>SQL92T</td>
<td>use JDBC SQLExceptions</td>
</tr>
<tr>
<td>Grouped operations</td>
<td>SQL92T</td>
<td>yes</td>
</tr>
<tr>
<td>Qualified * in select list</td>
<td>SQL92T</td>
<td>yes</td>
</tr>
<tr>
<td>Lowercase identifiers</td>
<td>SQL92T</td>
<td>yes</td>
</tr>
<tr>
<td>nullable PRIMARY KEYS</td>
<td>SQL92T</td>
<td>no</td>
</tr>
<tr>
<td><strong>Multiple schemas per user</strong></td>
<td>SQL92T</td>
<td>yes</td>
</tr>
<tr>
<td>SCHEMATA view</td>
<td></td>
<td>SYS.SYSSCHEMAS</td>
</tr>
<tr>
<td>Multiple module support</td>
<td>SQL92T</td>
<td>no (not required and not part of JDBC)</td>
</tr>
<tr>
<td>Referential delete actions</td>
<td>SQL92T</td>
<td>CASCADE, SET NULL, RESTRICT, and NO ACTION.</td>
</tr>
</tbody>
</table>
Table 8. Support for SQL-92 Features (continued)

<table>
<thead>
<tr>
<th>Feature</th>
<th>Source</th>
<th>Cloudscape 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAST functions</td>
<td>SQL92T</td>
<td>yes</td>
</tr>
<tr>
<td>INSERT expressions</td>
<td>SQL92T</td>
<td>yes</td>
</tr>
<tr>
<td>Explicit defaults</td>
<td>SQL92T</td>
<td>yes</td>
</tr>
<tr>
<td>Privilege tables</td>
<td>SQL92T</td>
<td>no</td>
</tr>
<tr>
<td>TABLE_PRIVILEGES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COLUMNS_PRIVILEGES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USAGE_PRIVILEGES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Keyword relaxations</td>
<td>SQL92T</td>
<td>yes</td>
</tr>
<tr>
<td>Domain definition</td>
<td>SQL92I</td>
<td>no</td>
</tr>
<tr>
<td>CASE expression</td>
<td>SQL92I</td>
<td>partial support</td>
</tr>
<tr>
<td>Compound character string constants</td>
<td>SQL92I</td>
<td>use concatenation</td>
</tr>
<tr>
<td>LIKE enhancements</td>
<td>SQL92I</td>
<td>yes</td>
</tr>
<tr>
<td>UNIQUE predicate</td>
<td>SQL92I</td>
<td>no</td>
</tr>
<tr>
<td>Table operations</td>
<td>SQL92I</td>
<td>yes</td>
</tr>
<tr>
<td>UNION relaxations</td>
<td>SQL92I</td>
<td>yes</td>
</tr>
<tr>
<td>EXCEPT</td>
<td></td>
<td>no</td>
</tr>
<tr>
<td>INTERSECT</td>
<td></td>
<td>no</td>
</tr>
<tr>
<td>CORRESPONDING</td>
<td></td>
<td>no</td>
</tr>
<tr>
<td>Schema definition statement</td>
<td>SQL92I</td>
<td>yes, partially</td>
</tr>
<tr>
<td>CREATE SCHEMA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>User authorization</td>
<td>SQL92I</td>
<td>use set schema</td>
</tr>
<tr>
<td>SET SESSION AUTHORIZATION</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CURRENT_USER</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>SESSION_USER</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>SYSTEM_USER</td>
<td></td>
<td>no</td>
</tr>
<tr>
<td>Constraint tables</td>
<td>SQL92I</td>
<td></td>
</tr>
<tr>
<td>TABLE_CONSTRAINTS</td>
<td>SYS.SYSCONSTRAINTS</td>
<td></td>
</tr>
<tr>
<td>REFERENTIAL_CONSTRAINTS</td>
<td>SYS.SYSFOREIGNKEYS</td>
<td></td>
</tr>
<tr>
<td>CHECK_CONSTRAINTS</td>
<td>SYS.SYSCHECKS</td>
<td></td>
</tr>
<tr>
<td>Usage tables</td>
<td>SQL92I</td>
<td></td>
</tr>
<tr>
<td>Intermediate information schema</td>
<td>SQL92I</td>
<td>use JDBC <code>DatabaseMetaData</code> and Cloudscape system tables</td>
</tr>
<tr>
<td>Subprogram support</td>
<td>SQL92I</td>
<td>not relevant to JDBC, which is much richer</td>
</tr>
<tr>
<td>Intermediate SQL Flagging</td>
<td>SQL92I</td>
<td>no</td>
</tr>
<tr>
<td>Schema manipulation</td>
<td>SQL92I</td>
<td>yes, to drop a default, set it to NULL.</td>
</tr>
<tr>
<td>Long identifiers</td>
<td>SQL92I</td>
<td>yes</td>
</tr>
<tr>
<td>Full outer join</td>
<td>SQL92I</td>
<td>no</td>
</tr>
<tr>
<td>Time zone specification</td>
<td>SQL92I</td>
<td>no</td>
</tr>
</tbody>
</table>
Table 8. Support for SQL-92 Features (continued)

<table>
<thead>
<tr>
<th>Feature</th>
<th>Source</th>
<th>Cloudscape 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scrolled cursors</td>
<td>SQL92I</td>
<td>partial (scrolling insensitive result sets through JDBC 2.0)</td>
</tr>
<tr>
<td>Intermediate set function support</td>
<td>SQL92I</td>
<td>partial</td>
</tr>
<tr>
<td>Character set definition</td>
<td>SQL92I</td>
<td>supports Java locales</td>
</tr>
<tr>
<td>Named character sets</td>
<td>SQL92I</td>
<td>supports Java locales</td>
</tr>
<tr>
<td>Scalar subquery values</td>
<td>SQL92I</td>
<td>yes</td>
</tr>
<tr>
<td>Expanded null predicate</td>
<td>SQL92I</td>
<td>yes</td>
</tr>
<tr>
<td>Constraint management</td>
<td>SQL92I</td>
<td>yes (ADD/DROP CONSTRAINT)</td>
</tr>
<tr>
<td>Documentation schema</td>
<td>SQL92I/FIPS 127-2</td>
<td>use JDBC DatabaseMetaData</td>
</tr>
<tr>
<td></td>
<td></td>
<td>use JDBC DatabaseMetaData</td>
</tr>
<tr>
<td>FOR BIT DATA types</td>
<td>SQL92F</td>
<td>yes</td>
</tr>
<tr>
<td>Assertion constraints</td>
<td>SQL92F</td>
<td>no</td>
</tr>
<tr>
<td>Temporary tables</td>
<td>SQL92F</td>
<td>IBM specific syntax only</td>
</tr>
<tr>
<td>Full dynamic SQL</td>
<td>SQL92F</td>
<td>no</td>
</tr>
<tr>
<td>Full DATETIME</td>
<td>SQL92F</td>
<td>no</td>
</tr>
<tr>
<td>Full DATETIME precision for TIME and TIMESTAMP</td>
<td>SQL92F</td>
<td>no</td>
</tr>
<tr>
<td>Full value expressions</td>
<td>SQL92F</td>
<td>yes</td>
</tr>
<tr>
<td>Truth value tests</td>
<td>SQL92F</td>
<td>yes</td>
</tr>
<tr>
<td>Full character functions</td>
<td>SQL92F</td>
<td>use Java methods or LOCATE</td>
</tr>
<tr>
<td>POSITION expression</td>
<td>SQL92F</td>
<td>use Java methods or LOCATE</td>
</tr>
<tr>
<td>UPPER/LOWER functions</td>
<td>SQL92F</td>
<td>yes</td>
</tr>
<tr>
<td>Derived tables in FROM</td>
<td>SQL92F</td>
<td>yes</td>
</tr>
<tr>
<td>Trailing underscore</td>
<td>SQL92F</td>
<td>yes</td>
</tr>
<tr>
<td>Indicator data types</td>
<td>SQL92F</td>
<td>not relevant to JDBC</td>
</tr>
<tr>
<td>Referential name order</td>
<td>SQL92F</td>
<td>no</td>
</tr>
<tr>
<td>Full SQL Flagging</td>
<td>SQL92F</td>
<td>no</td>
</tr>
<tr>
<td>Row and table constructors</td>
<td>SQL92F</td>
<td>yes</td>
</tr>
<tr>
<td>Catalog name qualifiers</td>
<td>SQL92F</td>
<td>no</td>
</tr>
<tr>
<td>Simple tables</td>
<td>SQL92F</td>
<td>no</td>
</tr>
<tr>
<td>Subqueries in CHECK</td>
<td>SQL92F</td>
<td>no, but can do with Java methods</td>
</tr>
<tr>
<td>Union join</td>
<td>SQL92F</td>
<td>no</td>
</tr>
<tr>
<td>Collation and translation</td>
<td>SQL92F</td>
<td>Java locales supported</td>
</tr>
<tr>
<td>Referential update actions</td>
<td>SQL92F</td>
<td>RESTRICT and NO ACTION. Can do others with triggers.</td>
</tr>
<tr>
<td>Feature</td>
<td>Source</td>
<td>Cloudscape 10</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>---------</td>
<td>------------------------</td>
</tr>
<tr>
<td>ALTER domain</td>
<td>SQL92F</td>
<td>no</td>
</tr>
<tr>
<td>INSERT column privileges</td>
<td>SQL92F</td>
<td>no</td>
</tr>
<tr>
<td>Referential MATCH types</td>
<td>SQL92F</td>
<td>no</td>
</tr>
<tr>
<td>View CHECK enhancements</td>
<td>SQL92F</td>
<td>no, views not updateable</td>
</tr>
<tr>
<td>Session management</td>
<td>SQL92F</td>
<td>use JDBC</td>
</tr>
<tr>
<td>Connection management</td>
<td>SQL92F</td>
<td>use JDBC</td>
</tr>
<tr>
<td>Self-referencing operations</td>
<td>SQL92F</td>
<td>yes</td>
</tr>
<tr>
<td>In-sensitive cursors</td>
<td>SQL92F</td>
<td>Yes through JDBC 2.0</td>
</tr>
<tr>
<td>Full set function</td>
<td>SQL92F</td>
<td>partially</td>
</tr>
<tr>
<td>Catalog flagging</td>
<td>SQL92F</td>
<td>no</td>
</tr>
<tr>
<td>Local table references</td>
<td>SQL92F</td>
<td>no</td>
</tr>
<tr>
<td>Full cursor update</td>
<td>SQL92F</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 8. Support for SQL-92 Features (continued)
Chapter 4. Cloudscape System Tables

Cloudscape includes the following system tables:
- "SYSALIASES"
- "SYSCHECKS" on page 194
- "SYSCOLUMNS" on page 194
- "SYSCONGLOMERATES" on page 195
- "SYSCONSTRAINTS" on page 195
- "SYSDEPENDS" on page 196
- "SYSFILES" on page 196
- "SYSFOREIGNKEYS" on page 197
- "SYSKEYS" on page 197
- "SYSSCHEMAS" on page 197
- "SYSTATISTICS" on page 197
- "SYSSTATEMENTS" on page 198
- "SYSTABLES" on page 198
- "SYSTRIGGERS" on page 199
- "SYSVIEWS" on page 200

You can query system tables, but you cannot alter them.

All system tables reside in the SYS schema. Because this is not the default schema, qualify all queries accessing the system tables with the SYS schema name.

The recommended way to get more information about these tables is to use an instance of the Java interface java.sql.DatabaseMetaData.

**SYSALIASES**

Describes the procedures and functions in the database.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullability</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALIASID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for the alias</td>
</tr>
<tr>
<td>ALIAS</td>
<td>VARCHAR</td>
<td>128</td>
<td>false</td>
<td>alias</td>
</tr>
<tr>
<td>SCHEMAID</td>
<td>CHAR</td>
<td>36</td>
<td>true</td>
<td>reserved for future use</td>
</tr>
<tr>
<td>JAVACLASSNAME</td>
<td>LONGVARCHAR</td>
<td>255</td>
<td>false</td>
<td>the Java class name</td>
</tr>
<tr>
<td>ALIATYPE</td>
<td>CHAR</td>
<td>1</td>
<td>false</td>
<td>'F' (function)/'P' (procedure)</td>
</tr>
<tr>
<td>NAMESPACE</td>
<td>CHAR</td>
<td>1</td>
<td>false</td>
<td>'F' (function)/'P' (procedure)</td>
</tr>
<tr>
<td>SYSTEMALIAS</td>
<td>BOOLEAN</td>
<td></td>
<td>false</td>
<td>true (system supplied or built-in alias)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>false (alias created by a user)</td>
</tr>
<tr>
<td>ALIASINFO</td>
<td>org.apache.derby. catalog.AliasInfo;</td>
<td></td>
<td>true</td>
<td>A Java interface that encapsulates the additional information that is specific to an alias</td>
</tr>
<tr>
<td></td>
<td>This class is not part of the public API</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**SYSCHECKS**

Describes the check constraints within the current database.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullability</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTRAINTID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for the constraint</td>
</tr>
<tr>
<td>CHECKDEFINITION</td>
<td>LONG VARCHAR</td>
<td></td>
<td>false</td>
<td>text of check constraint definition</td>
</tr>
<tr>
<td>REFERENCEDCOLUMNS</td>
<td>org.apache.derby.catalog.ReferencedColumns: This class is not part of the public API.</td>
<td>false</td>
<td>description of the columns referenced by the check constraint</td>
<td></td>
</tr>
</tbody>
</table>

**SYSCOLUMNS**

Describes the columns within all tables in the current database:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullable</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>REFERENCEID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>Identifier for table or publication (join with SYSTABLES.TABLEID or SYSPUBS.PUBLICATIONID)</td>
</tr>
<tr>
<td>COLUMN–NAME</td>
<td>CHAR</td>
<td>128</td>
<td>false</td>
<td>column or parameter name</td>
</tr>
<tr>
<td>COLUMNNUMBER</td>
<td>INT</td>
<td>4</td>
<td>false</td>
<td>the position of the column within the table or the position of the parameter within the publication</td>
</tr>
<tr>
<td>COLUMNDATATYPE</td>
<td>org.apache.derby.catalog.TypeDescriptor: This class is not part of the public API.</td>
<td>false</td>
<td>system type that describes precision, length, scale, nullability, type name, and storage type of data</td>
<td></td>
</tr>
<tr>
<td>COLUMNDEFAULT</td>
<td>java.io.Serializable</td>
<td></td>
<td>true</td>
<td>for tables, describes default value of the column. The toString() method on the object stored in the table returns the text of the default value as specified in the CREATE TABLE or ALTER TABLE statement. for publication parameters, defines the default value of the parameter at the source, and the actual value of the parameter at the target</td>
</tr>
<tr>
<td>COLUMNDEFAULTID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for the default value</td>
</tr>
<tr>
<td>IDENTITYCOLUMNVALUE</td>
<td>BIGINT</td>
<td></td>
<td>true</td>
<td>what the next value for column will be, if the column is an identity column</td>
</tr>
<tr>
<td>Column Name</td>
<td>Type</td>
<td>Length</td>
<td>Nullable</td>
<td>Contents</td>
</tr>
<tr>
<td>--------------------------</td>
<td>------------</td>
<td>--------</td>
<td>----------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>IDENTITYCOLUMNSTART</td>
<td>BIGINT</td>
<td></td>
<td>true</td>
<td>initial value of column (if specified), if it is an identity column</td>
</tr>
<tr>
<td>IDENTITYCOLUMNINC</td>
<td>BIGINT</td>
<td></td>
<td>true</td>
<td>amount column value is automatically incremented (if specified), if the column is an identity column</td>
</tr>
</tbody>
</table>

**SYSCONGLOMERATES**

Describes the conglomerates within the current database. A conglomerate is a unit of storage and is either a table or an index.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullable</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCHEMAID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>schema id for the conglomerate</td>
</tr>
<tr>
<td>TABLEID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>identifier for table (join with SYSTABLES.TABLEID)</td>
</tr>
<tr>
<td>CONGLOMERATENUMBER</td>
<td>BIGINT</td>
<td>8</td>
<td>false</td>
<td>conglomerate id for the conglomerate (heap or index)</td>
</tr>
<tr>
<td>CONGLOMERATENAME</td>
<td>VARCHAR</td>
<td>128</td>
<td>true</td>
<td>index name, if conglomerate is an index, otherwise the table ID</td>
</tr>
<tr>
<td>ISINDEX</td>
<td>BOOLEAN</td>
<td>1</td>
<td>false</td>
<td>whether or not conglomerate is an index</td>
</tr>
<tr>
<td>DESCRIPTOR</td>
<td>org.apache.derby.catalog.IndexDescriptor: This class is not part of the public API</td>
<td>true</td>
<td>system type describing the index</td>
<td></td>
</tr>
<tr>
<td>ISCONSTRAINT</td>
<td>BOOLEAN</td>
<td>1</td>
<td>false</td>
<td>whether or not conglomerate is a system-generated index enforcing a constraint</td>
</tr>
<tr>
<td>CONGLOMERATEID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for the conglomerate</td>
</tr>
</tbody>
</table>

**SYSCONSTRAINTS**

Describes the information common to all types of constraints within the current database (currently, this includes primary key, unique, foreign key, and check constraints).

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullable</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTRAINTID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for constraint</td>
</tr>
<tr>
<td>TABLEID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>identifier for table (join with SYSTABLES.TABLEID)</td>
</tr>
<tr>
<td>CONSTRAINT–NAME</td>
<td>VARCHAR</td>
<td>128</td>
<td>false</td>
<td>constraint name (internally generated if not specified by user)</td>
</tr>
</tbody>
</table>
### SYSDEPENDS

Describes the dependency relationships between persistent objects in the database. Persistent objects can be dependents (they depend on other objects) and/or providers (other objects depend on them).

Providers are tables, conglomerates, and constraints. Dependents are publications and views.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullable</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPENDENTID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for the dependent</td>
</tr>
<tr>
<td>DEPENDENTFINDER</td>
<td>org.apache.derby.catalog.DependableFinder:</td>
<td></td>
<td></td>
<td>system type describing the publication or view</td>
</tr>
<tr>
<td>PROVIDERID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for the provider</td>
</tr>
<tr>
<td>PROVIDERFINDER</td>
<td>org.apache.derby.catalog.DependableFinder:</td>
<td></td>
<td></td>
<td>system type describing the tables, conglomerates, and constraints that are providers</td>
</tr>
</tbody>
</table>

### SYSFILES

Describes jar files stored in the database.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullability</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILEID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for the jar file</td>
</tr>
<tr>
<td>SCHEMAID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>ID of the jar file’s schema (join with SYSSCHEMAS.SCHEMAID)</td>
</tr>
<tr>
<td>FILENAME</td>
<td>VARCHAR</td>
<td>128</td>
<td>false</td>
<td>SQL name of the jar file</td>
</tr>
<tr>
<td>GENERATIONID</td>
<td>BIGINT</td>
<td></td>
<td>false</td>
<td>Generation number for the file. When jar files are replaced, their generation identifiers are changed.</td>
</tr>
</tbody>
</table>
**SYSFOREIGNKEYS**

Describes the information specific to foreign key constraints in the current database.

Cloudscape generates a backing index for each foreign key constraint; the name of this index is the same as SYSFOREIGNKEYS.CONGLOMERATEID.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullability</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTRAINTID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for the foreign key constraint (join with SYSCONSTRAINTS.PERMISSIONID)</td>
</tr>
<tr>
<td>CONGLOMERATEID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for index backing up the foreign key constraint (join with SYSCONGLOMERATES.PERMISSIONID)</td>
</tr>
<tr>
<td>KEYCONSTRAINTID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for the primary key or unique constraint referenced by this foreign key (SYSKEYS.PERMISSIONID or SYSCONSTRAINTS.PERMISSIONID)</td>
</tr>
<tr>
<td>DELETERULE</td>
<td>CHAR</td>
<td>1</td>
<td>false</td>
<td>R for NO ACTION (default), S for RESTRICT, C for CASCADE, U for SET NULL</td>
</tr>
<tr>
<td>UPDATERULE</td>
<td>CHAR</td>
<td>1</td>
<td>false</td>
<td>R for NO ACTION (default), S for restrict</td>
</tr>
</tbody>
</table>

**SYSKEYS**

Describes the specific information for primary key and unique constraints within the current database. Cloudscape generates an index on the table to back up each such constraint. The index name is the same as SYSKEYS.CONGLOMERATEID.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullable</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTRAINTID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for constraint</td>
</tr>
<tr>
<td>CONGLOMERATEID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for backing index</td>
</tr>
</tbody>
</table>

**SYSSCHEMAS**

Describes the schemas within the current database.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullable</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCHEMAID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for the schema</td>
</tr>
<tr>
<td>SCHEMA–NAME</td>
<td>VARCHAR</td>
<td>128</td>
<td>false</td>
<td>schema name</td>
</tr>
<tr>
<td>AUTHORIZATIONID</td>
<td>VARCHAR</td>
<td>128</td>
<td>false</td>
<td>the authorization identifier of the owner of the schema</td>
</tr>
</tbody>
</table>

**SYSSTATISTICS**

Describes the schemas within the current database.
### Statistics

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullability</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for the statistic</td>
</tr>
<tr>
<td>REFERENCEID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>the conglomerate for which the statistic was created (join with SYSCONGLOMERATES, CONGLOMERATEID)</td>
</tr>
<tr>
<td>TABLEID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>the table for which the information is collected</td>
</tr>
<tr>
<td>CREATIONTIMESTAMP</td>
<td>TIMESTAMP</td>
<td></td>
<td></td>
<td>time when this statistic was created or updated</td>
</tr>
<tr>
<td>TYPE</td>
<td>CHAR</td>
<td>1</td>
<td>false</td>
<td>type of statistics</td>
</tr>
<tr>
<td>VALID</td>
<td>BOOLEAN</td>
<td></td>
<td>false</td>
<td>whether the statistic is still valid</td>
</tr>
<tr>
<td>COLCOUNT</td>
<td>INTEGER</td>
<td></td>
<td>false</td>
<td>number of columns in the statistic</td>
</tr>
<tr>
<td>STATISTICS</td>
<td>org.apache.derby.catalog.Statistics:</td>
<td></td>
<td>true</td>
<td>statistics information</td>
</tr>
</tbody>
</table>

### SYSSTATEMENTS

Contains one row per stored prepared statement.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullability</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>STMTID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for the statement</td>
</tr>
<tr>
<td>STMTNAME</td>
<td>VARCHAR</td>
<td>128</td>
<td>false</td>
<td>name of the statement</td>
</tr>
<tr>
<td>SCHEMAID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>the schema in which the statement resides</td>
</tr>
<tr>
<td>TYPE</td>
<td>CHAR</td>
<td>1</td>
<td>false</td>
<td>always 'S'</td>
</tr>
<tr>
<td>VALID</td>
<td>BOOLEAN</td>
<td></td>
<td>false</td>
<td>TRUE if valid, FALSE if invalid</td>
</tr>
<tr>
<td>TEXT</td>
<td>LONG VARCHAR</td>
<td></td>
<td>false</td>
<td>text of the statement</td>
</tr>
<tr>
<td>LASTCOMPILED</td>
<td>TIMESTAMP</td>
<td></td>
<td>true</td>
<td>time that the statement was compiled</td>
</tr>
<tr>
<td>COMPILATIONSCHEMAID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>id of the schema containing the statement</td>
</tr>
<tr>
<td>USINGTEXT</td>
<td>LONG VARCHAR</td>
<td></td>
<td>true</td>
<td>text of the USING clause of the CREATE STATEMENT and ALTER STATEMENT statements</td>
</tr>
</tbody>
</table>

### SYSTABLES

Describes the tables and views within the current database.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullable</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLEID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for table or view</td>
</tr>
</tbody>
</table>
### SYSTRIGGERS

Describes the database’s triggers.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullability</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRIGGERID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for the trigger</td>
</tr>
<tr>
<td>TRIGGERNAME</td>
<td>VARCHAR</td>
<td>128</td>
<td>false</td>
<td>name of the trigger</td>
</tr>
<tr>
<td>SCHEMAID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>id of the trigger’s schema (join with SYSSCHEMAS.SCHEMAID)</td>
</tr>
<tr>
<td>CREATIONTIMESTAMP</td>
<td>TIMESTAMP</td>
<td></td>
<td>false</td>
<td>time the trigger was created</td>
</tr>
<tr>
<td>EVENT</td>
<td>CHAR</td>
<td>1</td>
<td>false</td>
<td>‘U’ for update, ‘D’ for delete, ‘I’ for insert</td>
</tr>
<tr>
<td>FIRINGTIME</td>
<td>CHAR</td>
<td>1</td>
<td>false</td>
<td>‘B’ for before ‘A’ for after</td>
</tr>
<tr>
<td>TYPE</td>
<td>CHAR</td>
<td>1</td>
<td>false</td>
<td>‘R’ for row, ‘S’ for statement</td>
</tr>
<tr>
<td>STATE</td>
<td>CHAR</td>
<td>1</td>
<td>false</td>
<td>‘E’ for enabled, ‘D’ for disabled</td>
</tr>
<tr>
<td>TABLEID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>id of the table on which the trigger is defined</td>
</tr>
<tr>
<td>WHENSTMTID</td>
<td>CHAR</td>
<td>36</td>
<td>true</td>
<td>used only if there is a WHEN clause (not yet supported)</td>
</tr>
<tr>
<td>ACTIONSTMTID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>id of the stored prepared statement for the triggered–SQL–statement</td>
</tr>
<tr>
<td>REFERENCEDCOLUMNS</td>
<td>org.apache.derby.catalog.ReferencedColumns: This class is not part of the public API.</td>
<td></td>
<td>true</td>
<td>descriptor of the columns referenced by UPDATE triggers</td>
</tr>
<tr>
<td>TRIGGERDEFINITION</td>
<td>LONG VARCHAR</td>
<td></td>
<td>true</td>
<td>body of CREATE TRIGGER statement</td>
</tr>
<tr>
<td>REFERENCINGOLD</td>
<td>BOOLEAN</td>
<td></td>
<td>true</td>
<td>whether or not the REFERENCINGNAME, if non-null, refers to the OLD row or table</td>
</tr>
</tbody>
</table>
### REFERENCINGNEW
- **Type**: BOOLEAN
- **Length**: true
- **Nullability**: whether or not the REFERENCINGNAME, if non-null, refers to the NEW row or table

### REFERENCINGNAME
- **Type**: VARCHAR
- **Length**: 128
- **Nullability**: true
- **Contents**: the correlation name or identifier from the REFERENCING clause

Any SQL text that is part of a triggered–SQL–statement is compiled and stored in SYSSTATEMENTS. ACTIONSTMTID and WHENSTMTID are foreign keys that reference SYSSTATEMENTS.STMTID. The statements for a trigger are always in the same schema as the trigger.

### SYSVIEWS
Describes the view definitions within the current database.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Length</th>
<th>Nullability</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLEID</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>unique identifier for the view (called TABLEID since it is joined with column of that name in SYSTABLES)</td>
</tr>
<tr>
<td>VIEWDEFINITION</td>
<td>LONG VARCHAR</td>
<td></td>
<td>false</td>
<td>text of view definition</td>
</tr>
<tr>
<td>CHECKOPTION</td>
<td>CHAR</td>
<td>1</td>
<td>false</td>
<td>'N' (check option not supported yet)</td>
</tr>
<tr>
<td>COMPILATION</td>
<td>CHAR</td>
<td>36</td>
<td>false</td>
<td>id of the schema containing the view</td>
</tr>
<tr>
<td>SCHEMAID</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 5. Cloudscape Exception Messages and SQL States

- “Understanding SQLExceptions”
- “SQLState and Error Message Reference”

Understanding SQLExceptions

The JDBC driver returns SQLExceptions for all errors from Cloudscape. If the exception originated in a user type but is not itself an SQLException, it is wrapped in an SQLException. Cloudscape-specific SQLExceptions use SQLState class codes starting with X. Standard SQLState values are returned for exceptions where appropriate.

Cloudscape database exceptions are classified by severity. The severity of an SQLException is available through the getErrorCode method call on the SQLException. The severities are summarized below. For more information, check the javadoc for org.apache.derby.types.ExceptionSeverity:

- Warning Severity—a warning is given but current statement is completed.
- Statement Severity—the effects of the current statement, if any, on persistent data are undone.
- Transaction Severity—the effects of the current transaction on persistent data are undone; a rollback is performed.
- Session Severity—a rollback is performed and the current session is terminated. This closes the current connection.
- System Severity—the system is shut down. All uncommitted transactions are rolled back.

Unimplemented aspects of the JDBC driver return an SQLException with a message starting “Feature not implemented” and an SQLState of XJZZZ. These unimplemented parts are for features not supported by Cloudscape.

Cloudscape supplies values for the message and SQLState fields. In addition, Cloudscape sometimes returns multiple SQLExceptions using the nextException chain. The first exception is always the most severe exception, with SQL-92 Standard exceptions preceding those that are specific to Cloudscape.

For information on processing SQLExceptions, see “Working with Cloudscape SQLExceptions in an Application” in the IBM Cloudscape Developer’s Guide.

SQLState and Error Message Reference

- “Common Exceptions”

Common Exceptions

SQLStates for some common exceptions:

- deadlock
  40001
- foreign key violation
  23503
- check constraint violation
• duplicate value violating unique or primary key constraint
  23505
• duplicate value violating unique index
  23505
• truncation error
  22001
• user authorization error
  08004
• user authentication error (no permission to access database)
  04501
• user authentication error (no write access)
  22502
Chapter 6. JDBC Reference

Cloudscape comes with a built-in JDBC driver. That makes the JDBC API the only API for working with Cloudscape databases. The driver is a native protocol all-Java driver (type number four of types defined by Sun).

This chapter provides reference information about Cloudscape’s implementation of the JDBC API and documents the way it conforms to the JDBC 2.0 and 3.0 APIs.

See the IBM Cloudscape Developer’s Guide for task-oriented instructions on working with the driver.

This JDBC driver implements the standard JDBC interface defined by Sun. When invoked from an application running in the same JVM as Cloudscape, the JDBC driver supports connections to a Cloudscape database in embedded mode. No network transport is required to access the database. In client/server mode, the client application dispatches JDBC requests to the JDBC server over a network; the server, in turn, which runs in the same JVM as Cloudscape, sends requests to Cloudscape through the embedded JDBC driver.

The Cloudscape JDBC implementation provides access to Cloudscape databases and supplies all the required JDBC interfaces. Unimplemented aspects of the JDBC driver return an SQLException with a message stating “Feature not implemented” and an SQLState of XJZZZ. These unimplemented parts are for features not supported by Cloudscape.

- “Core JDBC java.sql Classes, Interfaces, and Methods”
- “JDBC 3.0-Only Features” on page 220

Core JDBC java.sql Classes, Interfaces, and Methods

This section details Cloudscape’s implementation of the following java.sql classes, interfaces, and methods:

- java.sql.Driver
- java.sql.DriverManager.getConnection
- java.sql.DriverManager.getPropertyInfo
- java.sql.Connection
- java.sql.DatabaseMetaData
- java.sql.Statement
- java.sql.PreparedStatement
- java.sql.CallableStatement
- java.sql.ResultSet
- java.sql.ResultSetMetaData
- java.sql.SQLException
- java.sql.SQLWarning
- java.sql.Types
**java.sql.Driver**

JDBC provides different ways to load a driver, listed below.

- "Embedded Databases"

**Embedded Databases**

The class that loads Cloudscape's local JDBC driver is the class `org.apache.derby.jdbc.EmbeddedDriver`. Some of the ways listed below create instances of the Cloudscape driver class. Do not use the class directly through the `java.sql.Driver` interface. Use the `DriverManager` class to create connections.

- `Class.forName("org.apache.derby.jdbc.EmbeddedDriver")`

Our recommended manner, because it ensures that the class is loaded in all JVMs by creating an instance at the same time.

- `new org.apache.derby.jdbc.EmbeddedDriver()`

Same as `Class.forName("org.apache.derby.jdbc.EmbeddedDriver")` except that it requires the class to be found when the code is compiled.

- `Class c = org.apache.derby.jdbc.EmbeddedDriver.class`

This is also the same as `Class.forName("org.apache.derby.jdbc.EmbeddedDriver")` except that it requires the class to be found when the code is compiled. The pseudo-static field `class` evaluates to the class that is named.

- Setting the System property `jdbc.drivers`

To set a System property, you alter the invocation command line or the system properties within your application. It is not possible to alter system properties within an applet.

An invocation of the IBM Application Developer Kit JVM:

```java
java -Djdbc.drivers=org.apache.derby.jdbc.EmbeddedDriver
applicationClass
```

The actual driver that gets registered in the `DriverManager` to handle the `jdbc:derby:` protocol is not the class `org.apache.derby.jdbc.EmbeddedDriver`; that class simply detects the type of Cloudscape driver needed and then causes the appropriate Cloudscape driver to be loaded.

The only supported way to connect to a Cloudscape system through the `jdbc:derby:` protocol is using the `DriverManager` to obtain a driver (`java.sql.Driver`) or connection (`java.sql.Connection`) through the `getDriver` and `getConnection` method calls.

**java.sql.DriverManager.getConnection**

A Java application using the JDBC API establishes a connection to a database by obtaining a `Connection` object. The standard way to obtain a `Connection` object is to call the method `DriverManager.getConnection`, which takes a String containing a database connection URL. A JDBC database connection URL (uniform resource locator) provides a way of identifying a database.

*DriverManager.getConnection* can take one argument besides a database connection URL, a `Properties` object. You can use the `Properties` object to set database connection URL attributes.

You can also supply strings representing user names and passwords. When they are supplied, Cloudscape checks whether they are valid for the current system if user authentication is enabled. User names are passed to Cloudscape as authorization identifiers, which are used to determine whether the user is
authorized for access to the database and for determining the default schema.
When the connection is established, if no user is supplied, Cloudscape sets the
default user to APP, which Cloudscape uses to name the default schema. If a user
is supplied, the default schema is the same as the user name.

Cloudscape Database Connection URL Syntax
A Cloudscape database connection URL consists of the basic database connection
URL followed by an optional subsubprotocol and optional attributes.

This section provides reference information only. For a more complete description,
including examples, see “Connecting to Databases” in Chapter 1 of the IBM
Cloudscape Developer’s Guide.

Embedded Databases
For applications with embedded databases, the syntax of the database connection
URL is

```
jdbc: derby: [subprotocol:] [databasesename]; [attributes]*
```

- **jdbc: derby:**
  In JDBC lingo, cloudscape is the subprotocol for connecting to a Cloudscape
database. The subprotocol is always cloudscape and does not vary.

- **subsubprotocol:**
  subsubprotocol, which is not typically specified, specifies where Cloudscape looks
  for a database: in a directory, in a class path, or in a jar file. It is used only in
  rare instances, usually for read-only databases. subsubprotocol is one of the
  following:
  - directory
  - classpath: Databases are treated as read-only databases, and all databaseNames
    must begin with at least a slash, because you specify them “relative” to the
    class path directory or archive. (You do not have to specify classpath as the
    subsubprotocol; it is implied.)
  - jar: Databases are treated as read-only databases.

  jar: requires an additional element immediately before the databaseName:
  - **pathToArchive**
    pathToArchive is the path to the jar or zip file that holds the database and
    includes the name of the jar or zip file.

  See the IBM Cloudscape Developer’s Guide for examples of database connection
  URLs for read-only databases.

- **databaseName**
  Specify the databaseName to connect to an existing database or a new one.
  You can specify the database name alone, or with a relative or absolute path. See
  “Standard Connections—Connecting to Databases in the File System” in Chapter
  1 of the IBM Cloudscape Developer’s Guide.

- **attributes**
  Specify 0 or more database connection URL attributes as detailed in [The
  Cloudscape Database Connection URL Attributes](#) on page 206.

Additional Syntax
Cloudscape also supports the following SQL standard syntax to obtain a reference
to the current connection in a database-side JDBC procedure or method:

```
jdbc: default: connection
```
The Cloudscape Database Connection URL Attributes

You can supply an optional list of attributes to a database connection URL. Cloudscape translates these attributes into properties, so you can also set attributes in a Properties object passed to DriverManager.getConnection. (You cannot set those attributes as system properties, only in an object passed to the DriverManager.getConnection method.)

These attributes are specific to Cloudscape and are listed in Chapter 7, “Database Connection URL Attributes,” on page 233.

Attribute name/value pairs are converted into properties and added to the properties provided in the connection call. If no properties are provided in the connection call, a properties set is created that contains only the properties obtained from the database connection URL.

```
Connection conn = DriverManager.getConnection(
    "jdbc:derby:sampleDB;create=true");
-- setting an attribute in a Properties object
Properties p = new Properties();
p.put("create", "true");
Connection conn = DriverManager.getConnection(  
    "jdbc:derby:sampleDB", myProps);
-- passing user name and password
Connection conn = DriverManager.getConnection(  
    "jdbc:derby:sampleDB", "dba", "password");
```

**Note:** Attributes are not parsed for correctness. If you pass in an incorrect attribute or corresponding value, it is simply ignored. (Cloudscape does provide a tool for parsing the correctness of attributes. For more information, see the IBM Cloudscape Tools and Utilities Guide.)

---

**java.sql.Driver.getPropertyInfo**

To get the DriverPropertyInfo object, request the JDBC driver from the driver manager:

```
java.sql.DriverManager.getDriver("jdbc:derby:").
getPropertyInfo(URL, Prop)
```

Do not request it from org.apache.derby.jdbc.EmbeddedDriver, which is only an intermediary class that loads the actual driver.

This method might return a DriverPropertyInfo object. In a Cloudscape system, it consists of an array of database connection URL attributes. The most useful attribute is **databaseName= nameofDatabase** which means that the object consists of a list of booted databases in the current system.

For example, if a Cloudscape system has the databases toursDB and flightsDB in its system directory, all the databases in the system are set to boot automatically, and a user has also connected to a database A:/dbs/tours94, the array returned from getPropertyInfo contains one object corresponding to the databaseName attribute. The choices field of the DriverPropertyInfo object will contain an array of three Strings with the values toursDB, flightsDB, and A:/dbs/tours94. Note that this object is returned only if the proposed connection objects do not already include a database name (in any form) or include the shutdown attribute with the value true.

For more information about java.sql.Driver.getPropertyInfo, see “Offering Connection Choices to the User” in Chapter 8 of the IBM Cloudscape Developer's Guide.
A Cloudscape Connection object is not garbage-collected until all other JDBC objects created from that connection are explicitly closed or are themselves garbage-collected. Once the connection is closed, no further JDBC requests can be made against objects created from the connection. Do not explicitly close the Connection object until you no longer need it for executing statements.

A session-severity or higher exception causes the connection to close and all other JDBC objects against it to be closed. System-severity exceptions cause the Cloudscape system to shut down, which not only closes the connection but means that no new connections should be created in the current JVM.

Changing the current isolation for the connection with setTransactionIsolation commits the current transaction and begins a new transaction, per the JDBC standard.

If you connect to a read-only database, the appropriate isReadOnly DatabaseMetaData value is returned. For example, Connections set to read-only using the setReadOnly method, Connections for which the user has been defined as a readOnlyAccess user (with one of the Cloudscape properties), and Connections to databases on read-only media return true.

Cloudscape does not use catalog names; the getCatalog and setCatalog methods result in a “Feature not implemented” SQLException with an SQLState of XJZZZ.

DatabaseMetaData result sets do not close the result sets of other statements, even when auto-commit is set to true.
DatabaseMetaData result sets are closed if a user performs any other action on a JDBC object that causes an automatic commit to occur. If you need the DatabaseMetaData result sets to be accessible while executing other actions that would cause automatic commits, turn off auto-commit with setAutoCommit(false).

**getProcedureColumns**

Cloudscape supports Java procedures. Cloudscape allows you to call Java procedures within SQL statements. Cloudscape returns information about the parameters in the getProcedureColumns call. If the corresponding Java method is overloaded, it returns information about each signature separately. Cloudscape returns information for all Java procedures defined by CREATE PROCEDURE.

getProcedureColumns returns a ResultSet. Each row describes a single parameter or return value.

**Parameters to getProcedureColumns**

The JDBC API defines the following parameters for this method call:
- **catalog**
  - always use null for this parameter in Cloudscape.
- **schemaPattern**
  - Java procedures have a schema.
- **procedureNamePattern**
  - a String object representing a procedure name pattern.
- **column–Name–Pattern**
  - a String object representing the name pattern of the parameter names or return value names. Java procedures have parameter names matching those defined in the CREATE PROCEDURE statement. Use “%” to find all parameter names.

**Columns in the ResultSet Returned by getProcedureColumns**

Columns in the ResultSet returned by getProcedureColumns are as described by the API. Further details for some specific columns:
- **PROCEDURE_CAT**
  - always "null" in Cloudscape
- **PROCEDURE_SCHEM**
  - schema for a Java procedure
- **PROCEDURE_NAME**
  - the name of the procedure
- **COLUMN_NAME**
  - the name of the parameter (see ["column–Name–Pattern"])
- **COLUMN_TYPE**
  - short indicating what the row describes. Always is DatabaseMetaData.procedureColumnIn for method parameters, unless the parameter is an array. If so, it is DatabaseMetaData.procedureColumnInOut. It always returns DatabaseMetaData.procedureColumnReturn for return values.
- **TYPE_NAME**
  - Cloudscape-specific name for the type.
- **NULLABLE**
  - always returns DatabaseMetaData.procedureNoNulls for primitive parameters and DatabaseMetaData.procedureNullable for object parameters
Remarks

A String describing the Java type of the method parameter

Method_ID

A Cloudscape-specific column.

DatabaseMetaData functionality not supported

In the current release, Cloudscape does not provide all of the DatabaseMetaData functionality. The following JDBC requests result in empty result sets, in the format required by the JDBC API:

- getColumnPrivileges
- getTablePrivileges

Cloudscape does not implement privileges, and thus has no information to provide for these calls.

getBestRowIdentifier looks for identifiers in this order:

- A primary key on the table
- A unique constraint or unique index on the table
- All the columns in the table

Because of this last choice, it will always find a set of columns that identify a row. However, if there are duplicate rows in the table, use of all columns might not necessarily identify a unique row in the table.

java.sql.Statement

- "ResultSet Objects"
- "java.sql.Statement functionality not supported" on page 210

ResultSet Objects

An error that occurs when a SELECT statement is first executed prevents a ResultSet object from being opened on it. The same error does not close the ResultSet if it occurs after the ResultSet has been opened.

For example, a divide-by-zero error that happens while the executeQuery method is called on a java.sql.Statement or java.sql.PreparedStatement throws an exception and returns no result set at all, while if the same error happens while the next method is called on a ResultSet object, it does not cause the result set to be closed.

Errors can happen when a ResultSet is first being created if the system partially executes the query before the first row is fetched. This can happen on any query that uses more than one table and on queries that use aggregates, GROUP BY, ORDER BY, DISTINCT, or UNION.

Closing a Statement causes all open ResultSet objects on that statement to be closed as well.

The cursor name for the cursor of a ResultSet can be set before the statement is executed. However, once it is executed, the cursor name cannot be altered.
**java.sql.Statement functionality not supported**

Cloudscape does not implement the following JDBC 1.2 methods of `java.sql.Statement`:

- `cancel`
- `setEscapeProcessing`
- `setQueryTimeout`

**java.sql.PreparedStatement**

Cloudscape provides all the required JDBC 1.2 type conversions and additionally allows use of the individual `setXXX` methods for each type as if a `setObject(Value, JDBCTypeCode)` invocation were made.

This means that `setString` can be used for any built-in target type.

The `setCursorName` method can be used on a `PreparedStatement` prior to an execute request to control the cursor name used when the cursor is created.

**Streaming Columns**

`setXXXStream` requests stream data between the application and the database.

JDBC allows an IN parameter to be set to a Java input stream for passing in large amounts of data in smaller chunks. When the statement is executed, the JDBC driver makes repeated calls to this input stream, reading its contents and transmitting those contents as the parameter data.

Cloudscape supports the three types of streams that JDBC 1.2 provides. These three streams are:

- `setBinaryStream`
  - for streams containing uninterpreted bytes
- `setAsciiStream`
  - for streams containing ASCII characters
- `setUnicodeStream`
  - for streams containing Unicode characters

JDBC requires that you specify the length of the stream. The stream object passed to these three methods can be either a standard Java stream object or the user’s own subclass that implements the standard `java.io.InputStream` interface.

According to the JDBC standard, streams can be stored only in columns of the data types shown in [Table 9](#). Streams cannot be stored in columns of the other built-in data types or of user-defined data types.

<table>
<thead>
<tr>
<th>Column Values</th>
<th>Type Correspondent</th>
<th>AsciiStream</th>
<th>UnicodeStream</th>
<th>BinaryStream</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOB</td>
<td><code>java.sql.Clob</code></td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>CHAR</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>VARCHAR</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>LONGVARCHAR</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>BINARY</td>
<td></td>
<td>x</td>
<td>x</td>
<td>X</td>
</tr>
</tbody>
</table>

[Table 9. Streamable JDBC Data Types](#)
Table 9. Streamable JDBC Data Types (continued)

<table>
<thead>
<tr>
<th>Column Values</th>
<th>Type Correspondent</th>
<th>AsciiStream</th>
<th>UnicodeStream</th>
<th>BinaryStream</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td>java.sql.Blob</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>VARBINARY</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>LONGVARBINARY</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>X</td>
</tr>
</tbody>
</table>

A large X indicates the preferred target data type for the type of stream. (See Table 11 on page 214.)

**Note:** If the stream is stored in a column of a type other than LONG VARCHAR or LONG VARCHAR FOR BIT DATA, the entire stream must be able to fit into memory at one time. Streams stored in LONG VARCHAR and LONG VARCHAR FOR BIT DATA columns do not have this limitation.

The following example shows how a user can store a streamed `java.io.File` in a LONG VARCHAR column:

```java
Statement s = conn.createStatement();
s.executeUpdate("CREATE TABLE atable (a INT, b LONG VARCHAR)"); conn.commit();
java.io.File file = new java.io.File("derby.txt");
int fileLength = (int) file.length();
// first, create an input stream
java.io.InputStream fin = new java.io.FileInputStream(file);
PreparedStatement ps = conn.prepareStatement("INSERT INTO atable VALUES (?, ?)");
ps.setInt(1, 1);
// set the value of the input parameter to the input stream
ps.setAsciiStream(2, fin, fileLength);
ps.execute();
conn.commit();
```

**java.sql.CallableStatement**

Cloudscape supports all the JDBC 1.2 methods of `CallableStatement`:
- `getBoolean()`
- `getByte()`
- `getBytes()`
- `getDate()`
- `getDouble()`
- `getFloat()`
- `getInt()`
- `getLong()`
- `getObject()`
- `getShort()`
- `getString()`
- `getTime()`
- `getTimestamp()`
- `registerOutParamter()`
- `wasNull()`
CallableStatements and OUT Parameters

Cloudscape supports OUT parameters and CALL statements that return values, as in the following example:

```
CallableStatement cs = conn.prepareCall(
    "? = CALL getDriverType(), CAST (? AS INT)"
);
cs.registerOutParameter(1, INT);
cs.setInt(2, 35);
cs.executeUpdate();
```

**Note:** Using a CALL statement with a procedure that returns a value is only supported with the `? =` syntax.

Register the output type of the parameter before executing the call.

CallableStatements and INOUT Parameters

INOUT parameters map to an array of the parameter type in Java. (The method must take an array as its parameter.) This conforms to the recommendations of the SQL standard.

Given the following example:

```
CallableStatement call = conn.prepareCall(
    "{CALL doubleMyInt(?)}"
);
// for inout parameters, it is good practice to
// register the outparameter before setting the input value
call.registerOutParameter(1, INTEGER);
call.setInt(1,10);
call.executeQuery();
int retval = call.getInt(1);
```

The method `doubleIt` should take a one-dimensional array of `ints`. Here is sample source code for that method:

```
public static void doubleMyInt(int[] i) {
    i[0] *=2;
    /* Cloudscape returns the first element of the array.*/
}
```

**Note:** The return value is *not* wrapped in an array even though the parameter to the method is.

<table>
<thead>
<tr>
<th>JDBC Type</th>
<th>Array Type for Method Parameter</th>
<th>Value and Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>long[]</td>
<td>long</td>
</tr>
<tr>
<td>BINARY</td>
<td>byte[][]</td>
<td>byte[]</td>
</tr>
<tr>
<td>BIT</td>
<td>boolean[]</td>
<td>boolean</td>
</tr>
<tr>
<td>DATE</td>
<td>java.sql.Date[]</td>
<td>java.sql.Date</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>double[]</td>
<td>double</td>
</tr>
<tr>
<td>FLOAT</td>
<td>double[]</td>
<td>double</td>
</tr>
<tr>
<td>INTEGER</td>
<td>int[]</td>
<td>int</td>
</tr>
<tr>
<td>LONGVARBINARY</td>
<td>byte[][]</td>
<td>byte[]</td>
</tr>
<tr>
<td>REAL</td>
<td>float[]</td>
<td>float</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>short[]</td>
<td>short</td>
</tr>
<tr>
<td>TIME</td>
<td>java.sql.Time[]</td>
<td>java.sql.Time</td>
</tr>
</tbody>
</table>
Table 10. INOUT Parameter Type Correspondence (continued)

<table>
<thead>
<tr>
<th>JDBC Type</th>
<th>Array Type for Method Parameter</th>
<th>Value and Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMESTAMP</td>
<td>java.sql.Timestamp[]</td>
<td>java.sql.Timestamp</td>
</tr>
<tr>
<td>VARBINARY</td>
<td>byte[][]</td>
<td>byte[]</td>
</tr>
<tr>
<td>OTHER</td>
<td>yourType[]</td>
<td>yourType</td>
</tr>
<tr>
<td>JAVA_OBJECT (only valid in Java2/JDBC 2.0 environments)</td>
<td>yourType[]</td>
<td>yourType</td>
</tr>
</tbody>
</table>

Register the output type of the parameter before executing the call. For INOUT parameters, it is good practice to register the output parameter before setting its input value.

java.sql.ResultSet

A positioned update or delete issued against a cursor being accessed through a ResultSet object modifies or deletes the current row of the ResultSet object.

Some intermediate protocols might pre-fetch rows. This causes positioned updates and deletes to operate against the row the underlying cursor is on, and not the current row of the ResultSet.

Cloudscape provides all the required JDBC 1.2 type conversions of the getXXX methods.

JDBC does not define the sort of rounding to use for ResultSet.getBigDecimal. Cloudscape uses java.math.BigDecimal.ROUND_HALF_DOWN.

Streaming Columns

If the underlying object is itself an OutputStream class, getBinaryStream returns the object directly.

To get a field from the ResultSet using streaming columns, you can use the getXXXStream methods if the type supports it. See Table 9 on page 210 for a list of types that support the various streams. (See also Table 11 on page 214.)

You can retrieve data from one of the supported data type columns as a stream, whether or not it was stored as a stream.

The following example shows how a user can retrieve a LONG VARCHAR column as a stream:

```java
// retrieve data as a stream
ResultSet rs = s.executeQuery("SELECT b FROM atable");
while (rs.next()) {
    // use an InputStream to get the data
    InputStream ip = rs.getAsciiStream(1);
    // process the stream--this is just a generic way to
    // print the data
    int c;
    int columnSize = 0;
    byte[] buff = new byte[128];
    for (;;) {
        int size = ip.read(buff);
        if (size == -1)
            break;
        ```
```
columnSize += size;
String chunk = new String(buff, 0, size);
    System.out.print(chunk);
} }
rs.close();
s.close();
conn.commit();

java.sql.ResultSetMetaData

- "Source and updatability (all data types)"

**Source and updatability (all data types)**
Cloudscape does not track the source or updatability of columns in ResultSets, and so always returns the following constants for the following methods:

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>isDefinitelyWritable</td>
<td>false</td>
</tr>
<tr>
<td>isReadOnly</td>
<td>false</td>
</tr>
<tr>
<td>isWritable</td>
<td>false</td>
</tr>
</tbody>
</table>

java.sql.SQLException

Cloudscape supplies values for the getMessage(), getSQLState(), and getErrorCode() calls of SQLExceptions. In addition, Cloudscape sometimes returns multiple SQLExceptions using the nextException chain. The first exception is always the most severe exception, with SQL-92 Standard exceptions preceding those that are specific to Cloudscape. For information on processing SQLExceptions, see “Working with Cloudscape SQLExceptions in an Application” in Chapter 5 of the IBM Cloudscape Developer’s Guide.

java.sql.SQLWarning

Cloudscape can generate a warning in certain circumstances. A warning is generated if, for example, you try to connect to a database with the create attribute set to true if the database already exists. Aggregates like sum() also raise a warning if NULL values are encountered during the evaluation.

All other informational messages are written to the Cloudscape system’s derby.log file.

java.sql.Types

Table 11 shows the mapping of java.sql.Types to SQL types.

**Table 11. Mapping of java.sql.Types to SQL Types**

<table>
<thead>
<tr>
<th>java.sql.Types</th>
<th>SQL Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>BINARY</td>
<td>CHAR FOR BIT DATA</td>
</tr>
<tr>
<td>BIT1</td>
<td>CHAR FOR BIT DATA</td>
</tr>
<tr>
<td>BLOB</td>
<td>BLOB (JDBC 2.0 and up)</td>
</tr>
<tr>
<td>CHAR</td>
<td>CHAR</td>
</tr>
</tbody>
</table>
Table 11. Mapping of java.sql.Types to SQL Types (continued)

<table>
<thead>
<tr>
<th>java.sql.Types</th>
<th>SQL Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOB</td>
<td>CLOB (JDBC 2.0 and up)</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>FLOAT</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>LONGVARBINARY</td>
<td>LONG VARCHAR FOR BIT DATA</td>
</tr>
<tr>
<td>LONGVARCHAR</td>
<td>LONG VARCHAR</td>
</tr>
<tr>
<td>NULL</td>
<td>Not a data type; always a value of a particular type</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>TIME</td>
<td>TIME</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>VARBINARY</td>
<td>VARCHAR FOR BIT DATA</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>VARCHAR</td>
</tr>
</tbody>
</table>

Notes:
1. BIT is only valid in JDBC 2.0 and earlier environments.
2. Values can be passed in using the FLOAT type code; however, these are stored as DOUBLE PRECISION values, and so always have the type code DOUBLE when retrieved.

**java.sql.Blob and java.sql.Clob**

In JDBC 2.0, java.sql.Blob is the mapping for the SQL BLOB (binary large object) type; java.sql.Clob is the mapping for the SQL CLOB (character large object) type.

java.sql.Blob and java.sql.Clob provide a logical pointer to the large object rather than a complete copy of the objects. Cloudscape processes only one data page into memory at a time. The whole BLOB does not need to be processed and stored in memory just to access the first few bytes of the LOB object.

Cloudscape now supports the built-in BLOB or CLOB data types. Cloudscape also provides the following support for these data types:

- **BLOB Features** Cloudscape supports the java.sql.Blob interface and the BLOB-related methods in java.sql.PreparedStatement and java.sql.ResultSet. The getBlob methods of CallableStatement are not implemented.

- **CLOB Features** Cloudscape supports the java.sql.Clob interface and the CLOB-related methods in java.sql.PreparedStatement and java.sql.ResultSet. The getClob methods of CallableStatement procedures are not implemented.

To use the java.sql.Blob and java.sql.Clob features:

- Use the SQL BLOB type for storage; LONG VARCHAR FOR BIT DATA, BINARY, and VARCHAR FOR BIT DATA types also work.
Use the SQL CLOB type for storage; LONG VARCHAR, CHAR, and VARCHAR types also work.

Use the getBlob or getBlob methods on the java.sql.ResultSet interface to retrieve a BLOB or CLOB handle to the underlying data.

You cannot call static methods (SQL extension over SQL) on any LOB—columns.

In addition, casting between strings and BLOBs is not recommended because casting is platform and database dependent.

Cloudscape uses unicode strings (2 byte characters), while other database products may use ASCII characters (1 byte per character). If various codpages are used, each character might need several bytes. A larger BLOB type might be necessary to accommodate a normal string in Cloudscape. You should use CLOB types for storing strings.

**Restrictions on BLOB, CLOB, (LOB-types):**

- LOB-types cannot be compared for equality(=) and non-equality(!=, <, >).
- LOB-typed values are not order-able, so <, <=, >, >= tests are not supported.
- LOB-types cannot be used in indices or as primary key columns.
- DISTINCT, GROUP BY, ORDER BY clauses are also prohibited on LOB-types.
- LOB-types cannot be involved in implicit casting as other base-types.

Cloudscape implements all of the methods for these JDBC 2.0 interfaces except for the set and get methods in CallableStatement interface.

**Recommendations:** Because the lifespan of a java.sql.Blob or java.sql.Clob ends when the transaction commits, turn off auto-commit with the java.sql.Blob or java.sql.Clob features.

### Table 12. JDBC 2.0 java.sql.Blob Methods Supported

<table>
<thead>
<tr>
<th>Returns</th>
<th>Signature</th>
<th>Implementation Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ByteArrayInputStream</td>
<td>getBinaryStream()</td>
<td></td>
</tr>
<tr>
<td>byte[]</td>
<td>getBytes(long pos, int length)</td>
<td>Exceptions are raised if pos &lt; 1, if pos is larger than the length of the , or if length &lt;= 0.</td>
</tr>
<tr>
<td>long</td>
<td>length()</td>
<td></td>
</tr>
<tr>
<td>long</td>
<td>position(byte[] pattern, long start)</td>
<td>Exceptions are raised if pattern == null, if start &lt; 1, or if pattern is an array of length 0.</td>
</tr>
<tr>
<td>long</td>
<td>position(Blob pattern, long start)</td>
<td>Exceptions are raised if pattern == null, if start &lt; 1, if pattern has length 0, or if an exception is thrown when trying to read the first byte of pattern.</td>
</tr>
</tbody>
</table>

### Table 13. JDBC 2.0 java.sql.Clob Methods Supported

<table>
<thead>
<tr>
<th>Returns</th>
<th>Signature</th>
<th>Implementation Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ByteArrayInputStream</td>
<td>getAsciiStream()</td>
<td></td>
</tr>
<tr>
<td>Reader</td>
<td>getCharacterStream()</td>
<td>NOT SUPPORTED</td>
</tr>
<tr>
<td>String</td>
<td>getSubString(long pos, int length)</td>
<td>Exceptions are raised if pos &lt; 1, if pos is larger than the length of the Clob, or if length &lt;= 0.</td>
</tr>
<tr>
<td>long</td>
<td>length()</td>
<td></td>
</tr>
</tbody>
</table>
Table 13. JDBC 2.0 java.sql.Clob Methods Supported (continued)

<table>
<thead>
<tr>
<th>Returns</th>
<th>Signature</th>
<th>Implementation Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>position(Clob searchstr, long start)</td>
<td>Exceptions are raised if searchStr == null or start &lt; 1, if searchStr has length 0, or if an exception is thrown when trying to read the first char of searchStr.</td>
</tr>
<tr>
<td>long</td>
<td>position(String searchstr, long start)</td>
<td>Exceptions are raised if searchStr == null or start &lt; 1, or if the pattern is an empty string.</td>
</tr>
</tbody>
</table>

Notes

The usual Cloudscape locking mechanisms (shared locks) prevent other transactions from updating or deleting the database item to which the java.sql.Blob or java.sql.Clob object is a pointer. However, in some cases, Cloudscape’s instantaneous lock mechanisms could allow a period of time in which the column underlying the java.sql.Blob or java.sql.Clob is unprotected. A subsequent call to getBlob/getClob, or to a java.sql.Blob/java.sql.Clob method, could cause undefined behavior.

Furthermore, there is nothing to prevent the transaction that holds the java.sql.Blob/java.sql.Clob (as opposed to another transaction) from updating the underlying row. (The same problem exists with the getXXXStream methods.) Program applications to prevent updates to the underlying object while a java.sql.Blob/java.sql.Clob is open on it; failing to do this could result in undefined behavior.

Do not call more than one of the ResultSet getXXX methods on the same column if one of the methods is one of the following:

- getBlob
- getClob
- getAsciiStream
- getBinaryStream
- getUnicodeStream

These methods share the same underlying stream; calling one more than one of these methods on the same column so could result in undefined behavior. For example:

```java
ResultSet rs = s.executeQuery("SELECT text FROM CLOBS WHERE i = 1");
while (rs.next()) {
    aclob=rs.getClob(1);
    ip = rs.getAsciiStream(1);
}
```

The streams that handle long-columns are not thread safe. This means that if a user chooses to open multiple threads and access the stream from each thread, the resulting behavior is undefined.

Clobes are not locale-sensitive.
### java.sql.Connection

#### Table 14. JDBC 2.0 Connection Methods Supported

<table>
<thead>
<tr>
<th>Returns</th>
<th>Signature</th>
<th>Implementation Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement</td>
<td>createStatement(int resultSetType, int resultSetConcurrency)</td>
<td>ResultSet.TYPE_FORWARD_ONLY and ResultSet.TYPE_SCROLL_INSENSITIVE are the only scrolling types supported. If you request TYPE_SCROLL_SENSITIVE, Cloudscape issues an SQLWarning and returns a TYPE_SCROLL_INSENSITIVE ResultSet.</td>
</tr>
<tr>
<td>PreparedStatement</td>
<td>prepareStatement(String sql, int resultSetType, int resultSetConcurrency)</td>
<td>ResultSet.CONCUR_READ_ONLY is the only concurrency supported. If you request a CONCUR_UPDATABLE ResultSet, Cloudscape issues an SQLWarning and returns a CONCUR_READ_ONLY ResultSet. (Use ResultSet.getWarnings to see warnings.)</td>
</tr>
<tr>
<td>CallableStatement</td>
<td>prepareCall(String sql, int resultSetType, int resultSetConcurrency)</td>
<td>ResultSet.TYPE_FORWARD_ONLY is the only resultSetType supported. ResultSet.CONCUR_READ_ONLY is the only concurrency supported. If you request a CONCUR_UPDATABLE ResultSet, Cloudscape issues an SQLWarning and returns a CONCUR_READ_ONLY ResultSet.</td>
</tr>
</tbody>
</table>

### java.sql.ResultSet

#### Table 15. JDBC 2.0 ResultSet Methods Supported

<table>
<thead>
<tr>
<th>Returns</th>
<th>Signature</th>
<th>Implementation Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>afterLast()</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>beforeFirst()</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>beforeFirst()</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>isFirst()</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>isAfterLast()</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>isBeforeFirst</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>isFirst()</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>isLast()</td>
<td>Always returns ResultSet.CONCUR_READ_ONLY. (Updatable ResultSets are not supported.)</td>
</tr>
<tr>
<td>int</td>
<td>getConcurrency()</td>
<td>Always returns 1.</td>
</tr>
<tr>
<td>int</td>
<td>getFetchDirection()</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>getFetchSize()</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>isAfterLast()</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>isBeforeFirst</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>isFirst()</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>isLast()</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>last()</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>previous()</td>
<td></td>
</tr>
</tbody>
</table>
Table 15. JDBC 2.0 ResultSet Methods Supported (continued)

<table>
<thead>
<tr>
<th>Returns</th>
<th>Signature</th>
<th>Implementation Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>relative(int rows)</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>setFetchDirection(int direction)</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>setFetchSize(int rows)</td>
<td>A fetch size of 1 is the only size supported.</td>
</tr>
</tbody>
</table>

Note: When working with scrolling insensitive ResultSets when auto-commit mode is turned on, the only positioning method that can close the ResultSet automatically is the next() method. When auto-commit mode is on, this method automatically closes the ResultSet if it is called and there are no more rows. afterLast() does not close the ResultSet, for example.

**java.sql.Statement**

Table 16. JDBC2.0 java.sql.Statement Methods Supported

<table>
<thead>
<tr>
<th>Returns</th>
<th>Signature</th>
<th>Implementation Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>addBatch(String sql)</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>clearBatch()</td>
<td></td>
</tr>
<tr>
<td>int[]</td>
<td>executeBatch()</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>getFetchDirection()</td>
<td>Method call does not throw an exception, but call is ignored.</td>
</tr>
<tr>
<td>int</td>
<td>getFetchSize()</td>
<td>Method call does not throw an exception, but call is ignored.</td>
</tr>
<tr>
<td>int</td>
<td>getMaxFieldSize()</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>getMaxRows()</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>setEscapeProcessing(boolean enable)</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>setFetchDirection(int direction)</td>
<td>Method call does not throw an exception, but call is ignored.</td>
</tr>
<tr>
<td>void</td>
<td>setFetchSize(int rows)</td>
<td>Method call does not throw an exception, but call is ignored.</td>
</tr>
<tr>
<td>void</td>
<td>setMaxFieldSize(int max)</td>
<td>Has no effect on Blobs and Clobs.</td>
</tr>
<tr>
<td>void</td>
<td>setMaxRows()</td>
<td></td>
</tr>
</tbody>
</table>

**java.sql.PreparedStatement**

Table 17. JDBC 2.0 java.sql.PreparedStatement Methods Supported

<table>
<thead>
<tr>
<th>Returns</th>
<th>Signature</th>
<th>Implementation Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>addBatch()</td>
<td></td>
</tr>
<tr>
<td>ResultSetMetaData</td>
<td>getMetaData()</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>setBlob(int i, Blob x)</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>setClob(int i, Clob x)</td>
<td></td>
</tr>
</tbody>
</table>
**java.sql.CallableStatement**

Table 18. JDBC 2.0 java.sql.CallableStatements Methods Supported

<table>
<thead>
<tr>
<th>Returns</th>
<th>Signature</th>
<th>Implementation Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>BigDecimal</td>
<td>getBigDecimal</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td>getDate(int, Calendar)</td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>getTime(int, Calendar)</td>
<td></td>
</tr>
<tr>
<td>Timestamp</td>
<td>getTimestamp(int, Calendar)</td>
<td></td>
</tr>
</tbody>
</table>

**java.sql.DatabaseMetaData**

Cloudscape implements all of the JDBC 2.0 methods for this interface.

**java.sql.ResultSetMetaData**

Cloudscape implements all of the JDBC 2.0 methods for this interface.

**java.sql.BatchUpdateException**

Thrown if there is a problem with a batch update.

**JDBC 3.0-Only Features**

JDBC 3.0 adds some functionality to the core API. This section documents the features supported by Cloudscape Version 10.

**Note:** These features are present only in a Java 2 version 1.4 or higher environment.

These features are:

- New DatabaseMetaData methods. See [“java.sql.DatabaseMetaData” on page 221](#).
- Retrieval of parameter metadata. See [“java.sql.PreparedStatement” on page 210](#) and [“java.sql.CommonsJavaSQLPreparedStatement” on page 210](#).
- Retrieval of auto-generated keys. See [“java.sql.Statement” on page 209](#) and [“java.sql.DatabaseMetaData” on page 221](#).
- Savepoints. See [“java.sql.Connection” on page 221](#).
- HOLD Cursors. See [“java.sql.DatabaseMetaData” on page 221](#).

The complete list:

- [“java.sql.Connection” on page 221](#)
- [“java.sql.DatabaseMetaData” on page 221](#)
- [“java.sql.ParameterMetaData” on page 221](#)
- [“java.sql.PreparedStatement” on page 210](#)
- [“java.sql.Savepoint” on page 222](#)
- [“java.sql.Statement” on page 209](#)
**java.sql.Connection**

Table 19. JDBC 3.0 Connection Methods Supported

<table>
<thead>
<tr>
<th>Returns</th>
<th>Signature</th>
<th>Implementation Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Savepoint</td>
<td>setSavepoint (String name)</td>
<td>Creates a savepoint with the given name in the current transaction and returns the new Savepoint object that represents it.</td>
</tr>
<tr>
<td>Savepoint</td>
<td>setSavepoint ()</td>
<td>Creates an unnamed savepoint in the current transaction and returns the new Savepoint object that represents it.</td>
</tr>
<tr>
<td>void</td>
<td>releaseSavepoint (Savepoint savepoint)</td>
<td>Removes the given Savepoint object from the current transaction.</td>
</tr>
<tr>
<td>void</td>
<td>rollback(Savepoint savepoint)</td>
<td>Undoes all changes made after the given Savepoint object was set.</td>
</tr>
</tbody>
</table>

**java.sql.DatabaseMetaData**

Table 20. JDBC 3.0 DatabaseMetaData Methods Supported

<table>
<thead>
<tr>
<th>Returns</th>
<th>Signature</th>
<th>Implementation Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>supportsSavepoints()</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>getDatabaseMajorVersion()</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>getDatabaseMinorVersion()</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>getJDBCMajorVersion()</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>getJDBCMinorVersion()</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>getSQLStateType()</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>supportsNamedParameters()</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>supportsMultipleOpenResults()</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>supportsGetGeneratedKeys()</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>supportsResultSetHoldability(int holdability)</td>
<td>returns ResultSet.HOLD_CURSORS_OVER_COMMIT</td>
</tr>
<tr>
<td>int</td>
<td>getResultSetHoldability()</td>
<td></td>
</tr>
</tbody>
</table>

**java.sql.ParameterMetaData**

ParameterMetaData is new in JDBC 3.0. It describes the number, type, and properties of parameters to prepared statements. The method PreparedStatement.getParameterMetaData returns a ParameterMetaData object that describes the parameter markers that appear in the PreparedStatement object. See "java.sql.PreparedStatement" on page 222 for more information.

Interface ParameterMetaData methods are listed below.

Table 21. JDBC 3.0 ParameterMetaData Methods

<table>
<thead>
<tr>
<th>Returns</th>
<th>Signature</th>
<th>Implementation Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>getCount()</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>isNullable(int param)</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>isSigned(int param)</td>
<td></td>
</tr>
</tbody>
</table>
Table 21. JDBC 3.0 ParameterMetaData Methods (continued)

<table>
<thead>
<tr>
<th>Returns</th>
<th>Signature</th>
<th>Implementation</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>getPrecision(int param)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>getScale(int param)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>getParameterType(int param)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>getParameterTypeName (int param)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>getParameterClassName (int param)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>getParameterMode (int param)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**java.sql.PreparedStatement**


Table 22. JDBC 3.0 PreparedStatement Methods

<table>
<thead>
<tr>
<th>Returns</th>
<th>Signature</th>
<th>Implementation</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParameterMetaData</td>
<td>getParameterMetaData()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**java.sql.Savepoint**

The `Savepoint` interface is new in JDBC 3.0. It contains new methods to set, release, or roll back a transaction to designated savepoints. Once a savepoint has been set, the transaction can be rolled back to that savepoint without affecting preceding work. Savepoints provide finer-grained control of transactions by marking intermediate points within a transaction.

**Setting and Rolling Back to a Savepoint**

The JDBC 3.0 API adds the method `Connection.setSavepoint`, which sets a savepoint within the current transaction. The `Connection.rollback` method has been overloaded to take a savepoint argument. See “[java.sql.Connection” on page 207 for more information.

The code example below inserts a row into a table, sets the savepoint `svpt1`, and then inserts a second row. When the transaction is later rolled back to `svpt1`, the second insertion is undone, but the first insertion remains intact. In other words, when the transaction is committed, only the row containing ‘1’ will be added to `TABLE1`.

```java
Statement stmt = conn.createStatement();
int rows = stmt.executeUpdate("INSERT INTO TABLE1 (COL1) VALUES(1)"); // set savepoint
Savepoint svpt1 = conn.setSavepoint("S1");
rows = stmt.executeUpdate("INSERT INTO TABLE1 (COL1) VALUES (2)");
...
conn.rollback(svpt1);
...
conn.commit();
```
Releasing a Savepoint

The method `Connection.releaseSavepoint` takes a Savepoint object as a parameter and removes it from the current transaction. Once a savepoint has been released, attempting to reference it in a rollback operation will cause an `SQLException` to be thrown.

Any savepoints that have been created in a transaction are automatically released and become invalid when the transaction is committed or when the entire transaction is rolled back.

Rolling a transaction back to a savepoint automatically releases and makes invalid any other savepoints created after the savepoint in question.

Rules

The savepoint cannot be set within a batch of statements to enabled partial recovery. If a savepoint is set any time before the method `executeBatch` is called, it is set before any of the statements that have been added to the batch are executed.

A savepoint-Name can be reused after it has been released explicitly (by issuing a release of savepoint) or implicitly (by issuing a connection commit/rollback).

Restrictions

Cloudscape does not support savepoints within a trigger. DB2 UDB has the same restriction.

Cloudscape does not release locks as part of the rollback to savepoint. DB2 UDB also retains all locks.

| Table 23. JDBC 3.0 Savepoint Methods

<table>
<thead>
<tr>
<th>Returns</th>
<th>Signature</th>
<th>Implementation Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td><code>getSavepointId()</code></td>
<td>Throws <code>SQLException</code> if this is a named savepoint.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Retrieves the generated ID for the savepoint that this Savepoint object represents.</td>
</tr>
<tr>
<td>String</td>
<td><code>getSavepointName()</code></td>
<td>Throws <code>SQLException</code> if this is an unnamed savepoint.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Retrieves the name of the savepoint that this Savepoint object represents.</td>
</tr>
</tbody>
</table>

java.sql.Statement

| Table 24. JDBC 3.0 Statement Methods

<table>
<thead>
<tr>
<th>Returns</th>
<th>Signature</th>
<th>Implementation Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResultSet</td>
<td><code>getGeneratedKeys()</code></td>
<td></td>
</tr>
</tbody>
</table>

Autogenerated Keys

JDBC 3.0’s autogenerated keys feature provides a way to retrieve values from columns that are part of an index or have a default value assigned. Cloudscape supports the autoincrement feature, which allows users to create columns in tables for which the database system automatically assigns increasing integer values. (See the IBM Cloudscape Reference Manual for more information.) In JDBC 3.0, the method `Statement.getGeneratedKeys` can be called to retrieve the value of such a
column. This method returns a ResultSet object with a column for each automatically generated key. Calling ResultSet.getMetaData on the ResultSet object returned by getGeneratedKeys produces a ResultSetMetaData object that can be used to determine the number, type, and properties of the generated keys. A flag indicating that any auto-generated columns should be returned is passed to the methods execute, executeUpdate, or prepareStatement when the statement is executed or prepared.

Here’s an example that returns a ResultSet with values for auto-generated columns in TABLE1:

```java
Statement stmt = conn.createStatement();
int rows = stmt.executeUpdate("INSERT INTO TABLE1 (C11, C12) VALUES (1,1)" , Statement.RETURN_GENERATED_KEYS);
ResultSet rs = stmt.getGeneratedKeys();
```

Additional methods allow you to specify the ordinals or names of the specific columns to be returned. An exception is thrown for invalid column or position names.

There are three ways of using Autogenerated Keys for insert statements. You can:

- Pass the flag Statement.RETURN_GENERATED_KEYS to execute or executeUpdate method.
- Send an array of column names to execute or executeUpdate, so only those column’s values are returned by getGeneratedKeys() resultset.
- Send an array of column indexes to execute or executeUpdate. This array is an index of columns for the target table.

If the Statement.RETURN_GENERATED_KEYS flag is passed to the execute or executeUpdate method, rather than the column positions/names list, Cloudscape returns a ResultSet containing columns with default values (this includes autoincrement column). To obtain a specific column, pass the column positions/names array. If Statement.getGeneratedKeys is executed for a non-insert statement, an exception is thrown.

The key indexes array in AutoGeneratedKey is an index of columns into the target table. You can send an array of column indexes to execute or executeUpdate. This array is an index of columns into the target table. For example:

```java
create table t1(c11 int, c12 int);
int[ ] colIndexes = new int[1];
colIndexes[0] = 1;
statement.executeUpdate("insert into t1(c12, c11) values (2,1)",colIndexes);
s.getGeneratedKeys();
--- will return a resultset with column c11.
```

See the IBM Cloudscape Developer’s Guide for more information about arrays.

**JDBC Escape Syntax**

JDBC provides a way of smoothing out some of the differences in the way different DBMS vendors implement SQL. This is called escape syntax. Escape syntax signals that the JDBC driver, which is provided by a particular vendor, scans for any escape syntax and converts it into the code that the particular database understands. This makes escape syntax DBMS-independent.

A JDBC escape clause begins and ends with curly braces. A keyword always follows the opening curly brace:

```
{keyword }
```
Cloudscape supports the following JDBC escape keywords, which are case-insensitive:

- **call**
  The escape keyword for use in CallableStatements.

- **d**
  The escape keyword for date formats.

- **escape**
  The keyword for specifying escape characters for LIKE clauses.

- **fn**
  The escape keyword for scalar functions.

- **oj**
  The escape keyword for outer joins.

- **t**
  The escape keyword for time formats.

- **ts**
  The escape keyword for timestamp formats.

Other JDBC escape keywords are not supported.

**Note:** Cloudscape returns the SQL unchanged in the Connection.nativeSQL call, since the escape syntax is native to SQL. In addition, it is unnecessary to call Statement.setEscapeProcessing for this reason.
call

This syntax is supported for a java.sql.Statement and a java.sql.PreparedStatement in addition to a CallableStatement.

Syntax

\{call statement \}

Example

-- Call a Java procedure
\{ call TOURS.BOOK_TOUR(?, ?) \}
Cloudscape interprets the JDBC escape syntax for date as equivalent to the SQL syntax for dates.

**Syntax**
{d \('yyyy-mm-dd'\)}

**Equivalent to**
DATE\('yyyy-mm-dd'\)

**Example**
VALUES {d \('1999-01-09'\)}
**escape**

The percent sign % and underscore _ are metacharacters within SQL LIKE clauses. JDBC provides syntax to force these characters to be interpreted literally. The JDBC clause immediately following a LIKE expression allows you to specify an escape character:

**Syntax**

```
WHERE CharacterExpression [ NOT ] LIKE
    CharacterExpressionWithWildCard
    { ESCAPE 'escapeCharacter' }
```

**Example**

```
-- find all rows in which a begins with the character "%"
SELECT a FROM tabA WHERE a LIKE '%$'
.escape '$'
-- find all rows in which a ends with the character "_"
SELECT a FROM tabA WHERE a LIKE '%=_'
.escape '='
```

**Note:** ? is not permitted as an escape character if the LIKE pattern is also a dynamic parameter (?).

In some languages, a single character consists of more than one collation unit (a 16-bit character). The escapeCharacter used in the escape clause must be a single collation unit in order to work properly.

You can also use the escape character sequence for LIKE without using JDBC’s curly braces; see "LIKE" on page 178.
The `fn` keyword allows the use of several scalar functions. The function name follows the keyword `fn`.

**Syntax**

```
{fn functionCall}
```

where `functionCall` is one of the following functions:

- `concat (CharacterExpression, CharacterExpression)`
- `sqrt (FloatingPointExpression)`
- `abs (NumericExpression)`
- `locate (CharacterExpression, CharacterExpression [, startIndex] )`
- `substring (CharacterExpression, startIndex, length)`

- `fn sqrt (FloatingPointExpression)`: is equivalent to built-in syntax `SQRT(FloatingPointExpression)` for more details see “SQRT” on page 126.
- `fn abs (NumericExpression)`: is equivalent to built-in syntax `ABSOLUTE(NumericExpression)` for more details see “ABS or ABSVAL” on page 86.
- `fn locate (CharacterExpression, CharacterExpression [, startIndex] )`: is equivalent to the built-in syntax `LOCATE(CharacterExpression, CharacterExpression [, StartPosition] )` for more details see “LOCATE” on page 114.
- `fn substring (CharacterExpression, startIndex, length)`: A character string formed by extracting `length` characters from the CharacterExpression beginning at `startIndex`; the index starts with 1.
- `fn mod (integer_type, integer_type)`: MOD returns the remainder (modulus) of argument 1 divided by argument 2. The result is negative only if argument 1 is negative. For more details, see “MOD” on page 120.

**Note:** Any Cloudscape built-in function is allowed in this syntax, not just those listed in this section.
Cloudscape interprets the JDBC escape syntax for outer joins (and all join operations) as equivalent to the correct SQL syntax for outer joins or the appropriate join operation.

For information about join operations, see "JOIN operation" on page 62.

Syntax
{oj \text{JOIN operation}[ \text{JOIN operation} ]* }

Equivalent to
\text{JOIN operation}[ \text{JOIN operation} ]*

Example
-- outer join
SELECT *
FROM {oj Countries LEFT OUTER JOIN Cities ON (Countries.country_ISO_code=Cities.country_ISO_code)}
-- another join operation
SELECT *
FROM {oj Countries JOIN Cities ON (Countries.country_ISO_code=Cities.country_ISO_code)}
-- you can have multiple join operations in a FROM clause
SELECT E.EMPNO, E.LASTNAME, M.EMPNO, M.LASTNAME
FROM {oj EMPLOYEE E INNER JOIN DEPARTMENT
INNER JOIN EMPLOYEE M ON MGRNO = M.EMPNO ON E.WORKDEPT = DEPTNO};
Cloudscape interprets the JDBC escape syntax for time as equivalent to the correct SQL syntax for times. Cloudscape also supports the ISO format of 8 characters (6 digits, and 2 decimal points).

**Syntax**

\{t 'hh:mm:ss'\}

**Equivalent to**

TIME 'hh:mm:ss'

**Example**

VALUES \{t '20:00:03'\}
Cloudscape interprets the JDBC escape syntax for timestamp as equivalent to the correct SQL syntax for timestamps. Cloudscape also supports the ISO format of 23 characters (17 digits, 3 dashes, and 3 decimal points).

**Syntax**

```sql
{ts 'yyyy-mm-dd hh:mm:ss.f...'}
```

**Equivalent to**

```sql
TIMESTAMP 'yyyy-mm-dd hh:mm:ss.f...'
```

The fractional portion of timestamp constants (f...) can be omitted.

**Example**

```sql
VALUES {ts '1999-01-09 20:11:11.123455'}
```
Chapter 7. Database Connection URL Attributes

Cloudscape allows you to supply a list of attributes to its database connection URL, which is a JDBC feature. The attributes are specific to Cloudscape.

You typically set attributes in a semicolon-separated list following the protocol and subprotocol. For information on how you set attributes, see “The Cloudscape Database Connection URL Attributes” on page 206. This chapter provides reference information only.

The core attributes of the Cloudscape database connection URL are:

- `bootPassword=key`
- `create=true`
- `databaseName=nameofDatabase`
- `dataEncryption=true`
- `encryptionProvider=providerName`
- `encryptionAlgorithm=algorithm`
- `territory=ll_CC`
- `logDevice=logDirectoryPath`
- `password=userPassword`
- `rollForwardRecoveryFrom=Path`
- `createFrom=Path`
- `restoreFrom=Path`
- `shutdown=true`
- `user=userName`
- “(no attributes)” on page 248

**Note:** Attributes are not parsed for correctness. If you pass in an incorrect attribute or corresponding value, it is simply ignored.
bootPassword=key

Function

Specifies the key to use for encrypting a new database or booting an existing encrypted database. Specify an alphanumeric string at least eight characters long.

Combining with Other Attributes

When creating a new database, must be combined with create=true and dataEncryption=true. When booting an existing encrypted database, no other attributes are necessary.

-- boot an encrypted database
jdbc:derby:encryptedDB;bootPassword=cseveryPlace

-- create a new, encrypted database
jdbc:derby:newDB;create=true;dataEncryption=true;
   bootPassword=cseveryPlace
create=true

**Function**

Creates the standard database specified within the database connection URL Cloudscape system and then connects to it. If the database cannot be created, the error appears in the error log and the connection attempt fails with an SQLException indicating that the database cannot be found.

If the database already exists, creates a connection to the existing database and an SQLWarning is issued.

JDBC does not remove the database on failure to connect at create time if failure occurs after the database call occurs. If a database connection URL used create=true and the connection fails to be created, check for the database directory. If it exists, remove it and its contents before the next attempt to create the database.

**Combining with Other Attributes**

You must specify a databaseName (after the subprotocol in the database connection URL) or a databaseName=nameofDatabase attribute with this attribute.

You can combine this attribute with other attributes. To specify a territory when creating a database, use the territory=ll_CC attribute.

**Note:** If you specify create=true and the database already exists, an SQLWarning is raised.

**Example**

```
jdbc:derby:sampleDB;create=true

jdbc:derby:;databaseName=newDB;create=true;
```
**databaseName=dbname**

**Function**

Specifies a database name for a connection; it can be used instead of specifying the database name in after the subprotocol.

For example, these URL (and Properties object) combinations are equivalent:

- `jdbc:derby:toursDB`
- `jdbc:derby:;databaseName=toursDB`
- `jdbc:derby:(with a property `databaseName` and its value set to `toursDB` in the `Properties` object passed into a connection request)`

If the database name is specified both in the URL (as a subname) and as an attribute, the database name set as the subname has priority. For example, the following database connection URL connects to `toursDB`:

```
jdbc:derby:toursDB;databaseName=flightsDB
```

Allowing the database name to be set as an attribute allows the `getPropertyInfo` method to return a list of choices for the database name based on the set of databases known to Cloudscape. For more information, see [java.sql.Driver.getPropertyInfo](#).

**Combining with Other Attributes**

You can combine this attribute with all other attributes.

**Example**

```
jdbc:derby:;databaseName=newDB;create=true
```
dataEncryption=true

**Function**

Specifies data encryption on disk for a new database. (For information about data encryption, see “Encrypting Databases on Disk” in the *IBM Cloudscape Developer’s Guide*.)

**Combining with Other Attributes**

Must be combined with `create=true` and `bootPassword=key`. You have the option of also specifying `encryptionProvider=providerName` and `encryptionAlgorithm=algorithm`.

**Example**

```
jdbc:derby:encryptedDB;create=true;dataEncryption=true;
  bootPassword=cLo4u922sc23aPe
```
encryptionProvider=providerName

**Function**

Specifies the provider for data encryption. (For information about data encryption, see “Encrypting Databases on Disk” in the *IBM Cloudscape Developer’s Guide*.)

If this attribute is not specified, the default encryption provider is `com.sun.crypto.provider.SunJCE`.

**Combining with Other Attributes**

Must be combined with `create=true`, `bootPassword=key`, and `dataEncryption=true`. You have the option of also specifying `encryptionAlgorithm=algorithm`.

**Example**

```sql
jdbc:derby:encryptedDB;create=true;dataEncryption=true;
    encryptionProvider=com.sun.crypto.provider.SunJCE;
    encryptionAlgorithm=DESede/CBC/NoPadding;
    bootPassword=cLo4u922sc23aPe
```
**encryptionAlgorithm=algorithm**

**Function**

Specifies the algorithm for data encryption.

Specify the algorithm per the Java conventions:

`algorithmName/feedbackMode/padding`

The only padding type allowed with Cloudscape is `NoPadding`.

If no encryption algorithm is specified, the default value is `DES/CBC/NoPadding`.

(For information about data encryption, see “Encrypting Databases on Disk” in Chapter 7 of the *IBM Cloudscape Developer’s Guide*).

**Combining with Other Attributes**

Must be combined with `create=true` `bootPassword=key` `dataEncryption=true` and `encryptionProvider=providerName`.

**Example**

```
jdbc:derby:encryptedDB;create=true;dataEncryption=true;  
   encryptionProvider=com.sun.crypto.provider.SunJCE;  
   encryptionAlgorithm=DESede/CBC/NoPadding;  
   bootPassword=cLo4u922sc23aPe
```

**Note:** If the specified provider does not support the specified algorithm, Cloudscape throws an exception.
Function

When creating or upgrading a database, use this attribute to associate a non-default territory with the database. Setting the `territory` attribute overrides the default system territory for that database. The default system territory is found using `java.util.Locale.getDefault()`.

Specify a territory in the form `ll_CC`, where `ll` is the two letter [Language Code] and `CC` is the two letter [Country Code].

Language codes consist of a pair of lower case letters that conform to ISO-639.

<table>
<thead>
<tr>
<th>Language Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>de</td>
<td>German</td>
</tr>
<tr>
<td>en</td>
<td>English</td>
</tr>
<tr>
<td>es</td>
<td>Spanish</td>
</tr>
<tr>
<td>ja</td>
<td>Japanese</td>
</tr>
</tbody>
</table>

Table 25. Sample Language Codes


Country codes consist of two uppercase letters that conform to ISO-3166.

<table>
<thead>
<tr>
<th>Country Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DE</td>
<td>Germany</td>
</tr>
<tr>
<td>US</td>
<td>United States</td>
</tr>
<tr>
<td>ES</td>
<td>Spain</td>
</tr>
<tr>
<td>MX</td>
<td>Mexico</td>
</tr>
<tr>
<td>JP</td>
<td>Japan</td>
</tr>
</tbody>
</table>

Table 26. Sample Country Codes


Combining with Other Attributes

The `territory` attribute is used only when creating a database.

Example

In the following example, the new database has a territory of Spanish language and Mexican nationality.

```
jdbc:derby:MexicanDB;create=true;territory=es_MX
```
**logDevice=logDirectoryPath**

**Function**

The `logDirectoryPath` specifies the path to the directory on which to store the database log during database creation or restore. Even if specified as a relative path, the `logDirectoryPath` is stored internally as an absolute path.

**Combining with Other Attributes**

Use in conjunction with `create=true` `createFrom`, `restoreFrom`, or `rollForwardRecoveryFrom`.

**Example**

```
jdbc:derby:newDB;create=true;logDevice=d:/newDBlog
```
password=userPassword

Function
A valid password for the given user name.

Combining with Other Attributes
Use in conjunction with user=userName.

Example
jdbc:derby:toursDB;user=jack;password=upTheHill
**rollForwardRecoveryFrom=Path**

**Function**

You can specify the `rollForwardRecoveryFrom=Path` in the boot time URL to restore the database using a backup copy and perform roll-forward recovery using archived and active logs.

To restore a database using roll-forward recovery, you must already have a backup copy of the database, all the archived logs since then, and the active log files. All the log files should be in the database log directory.

After a database is restored from full backup, transactions from the online archived logs and the active logs are replayed.

**Combining with Other Attributes**

Do not combine this attribute with `createFrom`, `restoreFrom`, or `create`.

**Example**

URL: `jdbc:derby:wombat;rollForwardRecoveryFrom=d:/backup/wombat`
createFrom=Path

Function
You can specify the createFrom=Path attribute in the boot time connection URL to create a database using a full backup at a specified location. If there is a database with the same name in Cloudscape.system.home, an error will occur and the existing database will be left intact. If there is not an existing database with the same name in the current Cloudscape.system.home location, the whole database is copied from the backup location to the Cloudscape.system.home location and started.

The Log files are copied to the default location. The logDevice attribute can be used in conjunction with createFrom=Path to store logs in a different location. With createFrom=Path you do not need to copy the individual log files to the log directory.

Combining with Other Attributes
Do not combine this attribute with rollforwardrecoveryFrom, restoreFrom, or create.

Example
URL: jdbc:derby:wombat;createFrom=d:/backup/wombat
**restoreFrom=Path**

**Function**

You can specify the `restoreFrom=Path` attribute in the boot time connection URL to restore a database using a full backup from the specified location. If a database with the same name exists in the Cloudscape.system.home location, the whole database is deleted, copied from the backup location, and then restarted.

The log files are copied to the same location they were in when the backup was taken. The logDevice attribute can be used in conjunction with `restoreFrom=Path` to store logs in a different location.

**Combining with Other Attributes**

Do not combine this attribute with createFrom, rollforwardrecoveryFrom, or create.

**Example**

URL: `jdbc:derby:wombat;restoreFrom=d:/backup/wombat`
shutdown=true

**Function**

Shuts down the specified database if you specify a `databaseName`. (Reconnecting to the database reboots the database.)

Shuts down the entire Cloudscape system if and only if you do not specify a `databaseName`.

When you are shutting down a single database, it lets Cloudscape perform a final checkpoint on the database.

When you are shutting down a system, it lets Cloudscape perform a final checkpoint on all system databases, deregister the JDBC driver, and shut down within the JVM before the JVM exits. A successful shutdown always results in an `SQLException` indicating that Cloudscape has shut down and that there is no connection. Once Cloudscape is shut down, you can restart it by reloading the driver. For details on restarting Cloudscape, see “Shutting Down the System” in Chapter 1 of the *IBM Cloudscape Developer’s Guide*.

*Checkpointing* means writing all data and transaction information to disk so that no recovery needs to be performed at the next connection.

Used to shut down the entire system only when it is embedded in an application.

**Note:** Any request to the `DriverManager` with a `shutdown=true` attribute raises an exception.

**Example**

```bash
-- shuts down entire system
jdbc:derby:;shutdown=true
-- shuts down salesDB
jdbc:derby:salesDB;shutdown=true
```
user=userName

Specifies a valid user name for the system, specified with a password. A valid user name and password are required when user authentication is turned on.

**Combining with Other Attributes**

Use in conjunction with `password=userPassword`.

**Example**

The following database connection URL connects the user jill to `toursDB`:

```java
jdbc:derby:toursDB;user=jill;password=toFetchAPail
```
If no attributes are specified, you must specify a databaseName.

Cloudscape opens a connection to an existing database with that name in the current system directory. If the database does not exist, the connection attempt returns an SQLException indicating that the database cannot be found.

Example

jdbc:derby:mydb
Chapter 8. J2EE Compliance: Java Transaction API and javax.sql Extensions

- “J2EE Overview”
- “The JTA API” on page 250
- “javax.sql: JDBC Extensions” on page 251

J2EE Overview

J2EE, or the Java 2 Platform, Enterprise Edition, is a standard for development of enterprise applications; one example is Enterprise Java Beans (EJBs) with distributed capabilities.

Cloudscape Version 10 is a J2EE-conformant component in a distributed J2EE system. As such, it is one part of a larger system that includes, among other things, a JNDI server, a connection pool module, a transaction manager, a resource manager, and user applications. (Cloudscape also supports the JTA API, which is not a current J2EE requirement, but this functionality provides another piece of the same system.) Within this system, Cloudscape can serve as the resource manager.

For more information on J2EE, see the J2EE specification available at http://java.sun.com/j2ee/docs.html.

In order to qualify as a resource manager in a J2EE system, J2EE requires these basic areas of support:

- JNDI support.
  Allows calling applications to register names for databases and access them through those names instead of through database connection URLs. Implementation of one of the JDBC extensions, javax.sql.DataSource, provides this support.

- Connection pooling
  A mechanism by which a connection pool server keeps a set of open connections to a resource manager (Cloudscape). A user requesting a connection can get one of the available connections from the pool. Such a connection pool is useful in client/server environments because establishing a connection is relatively expensive. In an embedded environment, connections are much cheaper, making the performance advantage of a connection pool negligible. Implementation of two of the JDBC extensions, javax.sql.ConnectionPoolDataSource and javax.sql.PooledConnection, provide this support.

- XA support.
  XA is one of several standards for distributed transaction management. It is based on two-phase commit. The javax.sql.XAxxx interfaces, along with java.transaction.xa package, are an abstract implementation of XA. For more information about XA, see X/Open CAE Specification—Distributed Transaction Processing: The XA Specification, X/Open Document No. XO/CAE/91/300 or ISBN 1 872630 24 3. Implementation of the JTA API, the interfaces of the java.transaction.xa package, [javax.sql.XAConnection, javax.sql.XADataSource, javax.transaction.xa.XAResource, javax.transaction.xa.Xid, and javax.transaction.xa.XAException], provide this support.
With the exception of the core JDBC interfaces, these interfaces are not visible to the end-user application; instead, they are used only by the other back-end components in the system.

Note: For information on the classes that implement these interfaces and how to use Cloudscape as a resource manager, see Chapter 6, “Using Cloudscape as a J2EE Resource Manager” in the IBM Cloudscape Developer’s Guide.

**JVM and Libraries for J2EE Features**

These features require the following:

- Java™ 2 Platform, Standard Edition v 1.2 (J2SE) environment or greater
- `javax.sql` libraries
  
  The JDBC 2.0 standard extension binaries are available from [http://java.sun.com/products/jdbc/download.html](http://java.sun.com/products/jdbc/download.html) These libraries are part of the standard environment from Java™ 2 Platform, Standard Edition v 1.4 or later.
- `javax.transaction.xa` libraries
  
  These libraries are part of the standard environment from Java™ 2 Platform, Standard Edition v 1.4 or later.
  
- Cloudscape (derby.jar)

---

**The JTA API**

The JTA API is made up of the two interfaces and one exception that are part of the `javax.transaction.xa` package. Cloudscape fully implements this API.

- `javax.transaction.xa.XAResource`
- `javax.transaction.xa.Xid`
- `javax.transaction.xa.XAException`

---

**Notes on Product Behavior**

**Recovered Global Transactions**

Using the `XAResource.prepare` call causes a global transaction to enter a prepared state, which allows it to be persistent. Typically, the prepared state is just a transitional state before the transaction outcome is determined. However, if the system crashes, recovery puts transactions in the prepared state back into that state and awaits instructions from the transaction manager.

**XAConnections and User Names and Passwords**

If a user opens an `XAConnection` with a user name and password, the transaction it created cannot be attached to an `XAConnection` opened with a different user name and password. A transaction created with an `XAConnection` without a user name and password can be attached to any `XAConnection`.

However, the user name and password for recovered global transactions are lost; any `XAConnection` can commit or roll back that in-doubt transaction.

Note: Cloudscape’s `XADatasources` can only be embedded. No remote (client/server) support is provided.
javax.sql: JDBC Extensions

This section documents the JDBC extensions that Cloudscape implements for J2EE compliance. (For more details about these extensions, see [http://java.sun.com/products/jdbc/jdbc20.stdext.javadoc/javax/sql/package-summary.html](http://java.sun.com/products/jdbc/jdbc20.stdext.javadoc/javax/sql/package-summary.html).)

- `javax.sql.DataSource`
  Cloudscape’s implementation of `DataSource` means that it supports JNDI; as a resource manager, it allows a database to be named and registered within a JNDI server. This allows the calling application to access the database by a name (as a data source) instead of through a database connection URL.

- `javax.sql.ConnectionPoolDataSource` and `javax.sql.PooledConnection`
  Establishing a connection to the database can be a relatively expensive operation in client/server environments. Establishing the connection once and then using the same connection for multiple requests can dramatically improve the performance of a database.
  
The Cloudscape implementation of `ConnectionPoolDataSource` and `PooledConnection` allows a connection pool server to maintain a set of such connections to the resource manager (Cloudscape). In an embedded environment, connections are much cheaper and connection pooling is not necessary.

- `javax.sql.XAConnection`
  An `XAConnection` produces an `XAResource`, and, over its lifetime, many `Connections`. It allows for distributed transactions.

- `javax.sql.XADataSource`
  An `XADataSource` is simply a `ConnectionPoolDataSource` that produces `XAConnnections`.

In addition, Cloudscape provides three methods for `XADataSource`, `DataSource`, and `ConnectionPoolDataSource`. Cloudscape supports a number of additional data source properties:

- `setCreateDatabase(String create)`
  Sets a property to create a database at the next connection. The string argument must be “create”.

- `setShutdownDatabase(String shutdown)`
  Sets a property to shut down a database. Shuts down the database at the next connection. The string argument must be “shutdown”.

**Note:** Set these properties before getting the connection.
Appendix A. Cloudscape API

Cloudscape provides Javadoc HTML files of API classes and interfaces in the javadoc subdirectory. This appendix provides a brief overview of the API. Cloudscape does not provide the Javadoc for the java.sql packages, the main API for working with Cloudscape, because that is part of the IBM Application Developer Kit. For information about Cloudscape’s implementation of JDBC, see Chapter 6, “JDBC Reference,” on page 203.

This document divides the API classes and interfaces into three categories:

- “Stand-Alone Tools and Utilities”
  These are Java classes that stand on their own and are invoked in a command window.

- “[JDBC Implementation Classes”
  You do not invoke these on the command-line, only within a specified context within another application. These classes are standard JDBC APIs.

Stand-Alone Tools and Utilities

These classes are in the packages org.apache.derby.tools.

- org.apache.derby.tools.ij
  An SQL scripting tool that can run as an embedded or a remote client/server application. See the IBM Cloudscape Tools and Utilities Guide.

- org.apache.derby.tools.sysinfo
  A command-line, server-side utility that displays information about your JVM and Cloudscape product. See the IBM Cloudscape Tools and Utilities Guide.

- org.apache.derby.tools.dblook
  A utility to view all or parts of the Data Definition Language (DDL) for a given database. See the IBM Cloudscape Tools and Utilities Guide.

JDBC Implementation Classes

JDBC Driver

This is the JDBC driver for Cloudscape:

- org.apache.derby.jdbc.EmbeddedDriver
  Used to boot the embedded built-in JDBC driver and the Cloudscape system. See the IBM Cloudscape Developer’s Guide.

Data Source Classes

These classes are all related to Cloudscape’s implementation of javax.sql.DataSource and related APIs. For more information, see Chapter 6, “Using Cloudscape as a J2EE Resource Manager” IBM Cloudscape Developer’s Guide.

- org.apache.derby.jdbc.EmbeddedDataSource
- org.apache.derby.jdbc.EmbeddedConnectionPoolDataSource
- org.apache.derby.jdbc.EmbeddedXADataSource
Miscellaneous Utilities and Interfaces

- `org.apache.derby.authentication.UserAuthenticator`
- An interface provided by Cloudscape. Classes that provide an alternate user authentication scheme must implement this interface. For information about users, see “Working with User Authentication” in Chapter 7 of the IBM Cloudscape Developer’s Guide.
Appendix B. Supported territories

The following is a list of supported territories:

<table>
<thead>
<tr>
<th>Territory</th>
<th>Cloudscape territory setting (derby.territory)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Japanese</td>
<td>ja_JP</td>
</tr>
<tr>
<td>Korean</td>
<td>ko_KR</td>
</tr>
<tr>
<td>Chinese (Traditional)</td>
<td>zh_TW</td>
</tr>
<tr>
<td>Chinese (Simplified)</td>
<td>zh_CN</td>
</tr>
<tr>
<td>French</td>
<td>fr</td>
</tr>
<tr>
<td>German</td>
<td>de_DE</td>
</tr>
<tr>
<td>Italian</td>
<td>it</td>
</tr>
<tr>
<td>Spanish</td>
<td>es</td>
</tr>
<tr>
<td>Portuguese (Brazilian)</td>
<td>pt_BR</td>
</tr>
</tbody>
</table>
Appendix C. Cloudscape limitations

The following tables list the limitations associated with Cloudscape. They include:

- “Identifier length limitations”
- “Numeric limitations”
- “String limitations” on page 258
- “DATE, TIME, and TIMESTAMP limitations” on page 258
- “Database Manager limitations” on page 259

Identifier length limitations

Table 27. Identifier length limitations. The following table contains limitations on identifier lengths in Cloudscape.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Maximum number of characters allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraint name</td>
<td>18</td>
</tr>
<tr>
<td>correlation name</td>
<td>128</td>
</tr>
<tr>
<td>cursor name</td>
<td>18</td>
</tr>
<tr>
<td>data source column name</td>
<td>128</td>
</tr>
<tr>
<td>data source index name</td>
<td>128</td>
</tr>
<tr>
<td>data source name</td>
<td>128</td>
</tr>
<tr>
<td>savepoint name</td>
<td>128</td>
</tr>
<tr>
<td>schema name</td>
<td>30</td>
</tr>
<tr>
<td>unqualified column name</td>
<td>30</td>
</tr>
<tr>
<td>unqualified function name</td>
<td>18</td>
</tr>
<tr>
<td>unqualified index name</td>
<td>18</td>
</tr>
<tr>
<td>unqualified procedure name</td>
<td>128</td>
</tr>
<tr>
<td>parameter name</td>
<td>128</td>
</tr>
<tr>
<td>unqualified trigger name, index name</td>
<td>18</td>
</tr>
<tr>
<td>unqualified table name, view name, stored procedure name</td>
<td>128</td>
</tr>
</tbody>
</table>

Numeric limitations

Table 28. Numeric limitations. The following table contains limitations on numeric values in Cloudscape.

<table>
<thead>
<tr>
<th>Value</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smallest INTEGER</td>
<td>-2,147,483,648</td>
</tr>
<tr>
<td>Largest INTEGER</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>Smallest BIGINT</td>
<td>-9,223,372,036,854,775,808</td>
</tr>
<tr>
<td>Largest BIGINT</td>
<td>9,223,372,036,854,775,807</td>
</tr>
<tr>
<td>Smallest SMALLINT</td>
<td>-32,768</td>
</tr>
<tr>
<td>Largest SMALLINT</td>
<td>32,767</td>
</tr>
</tbody>
</table>
Table 28. Numeric limitations (continued). The following table contains limitations on numeric values in Cloudscape.

<table>
<thead>
<tr>
<th>Value</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Largest decimal precision</td>
<td>31,255</td>
</tr>
<tr>
<td>Smallest DOUBLE</td>
<td>-1.79769E+308</td>
</tr>
<tr>
<td>Largest DOUBLE</td>
<td>1.79769E+308</td>
</tr>
<tr>
<td>Smallest positive DOUBLE</td>
<td>2.225E-307</td>
</tr>
<tr>
<td>Largest negative DOUBLE</td>
<td>-2.225E-307</td>
</tr>
<tr>
<td>Smallest REAL</td>
<td>-3.402E+38</td>
</tr>
<tr>
<td>Largest REAL</td>
<td>3.402E+38</td>
</tr>
<tr>
<td>Smallest positive REAL</td>
<td>1.175E-37</td>
</tr>
<tr>
<td>Largest negative REAL</td>
<td>-1.175E-37</td>
</tr>
</tbody>
</table>

String limitations

Table 29. String limitations. The following table contains limitations on string values in Cloudscape.

<table>
<thead>
<tr>
<th>Value</th>
<th>Maximum Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length of CHAR</td>
<td>254 characters</td>
</tr>
<tr>
<td>Length of VARCHAR</td>
<td>32,672 characters</td>
</tr>
<tr>
<td>Length of LONG VARCHAR</td>
<td>32,700 characters</td>
</tr>
<tr>
<td>Length of CLOB</td>
<td>2,147,483,647 characters</td>
</tr>
<tr>
<td>Length of BLOB</td>
<td>2,147,483,647 characters</td>
</tr>
<tr>
<td>Length of character constant</td>
<td>32,672</td>
</tr>
<tr>
<td>Length of concatenated character string</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>Length of concatenated binary string</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>Number of hex constant digits</td>
<td>16,336</td>
</tr>
<tr>
<td>Length of DOUBLE value constant</td>
<td>30 characters</td>
</tr>
</tbody>
</table>

DATE, TIME, and TIMESTAMP limitations

Table 30. DATE, TIME, and TIMESTAMP limitations. The following table contains limitations on date, time, and timestamp values in Cloudscape.

<table>
<thead>
<tr>
<th>Value</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smallest DATE value</td>
<td>0001-01-01</td>
</tr>
<tr>
<td>Largest DATE value</td>
<td>9999-12-31</td>
</tr>
<tr>
<td>Smallest TIME value</td>
<td>00:00:00</td>
</tr>
<tr>
<td>Largest TIME value</td>
<td>24:00:00</td>
</tr>
<tr>
<td>Smallest TIMESTAMP value</td>
<td>0001-01-01-00.00.00.0000000000</td>
</tr>
<tr>
<td>Largest TIMESTAMP value</td>
<td>9999-12-31-24.00.00.0000000000</td>
</tr>
</tbody>
</table>
# Database Manager limitations

*Table 31. Database manager limitations.* The following table contains limitations on various Database Manager values in Cloudscape.

<table>
<thead>
<tr>
<th>Value</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Most columns in a table</td>
<td>1,012</td>
</tr>
<tr>
<td>Most columns in a view</td>
<td>5,000</td>
</tr>
<tr>
<td>Maximum number of parameters in a stored procedure</td>
<td>90</td>
</tr>
<tr>
<td>Most indexes on a table</td>
<td>32,767 or storage capacity</td>
</tr>
<tr>
<td>Most tables referenced in an SQL statement or a view</td>
<td>storage capacity</td>
</tr>
<tr>
<td>Most elements in a select list</td>
<td>1,012</td>
</tr>
<tr>
<td>Most predicates in a WHERE or HAVING clause</td>
<td>storage capacity</td>
</tr>
<tr>
<td>Maximum number of columns in a GROUP BY clause</td>
<td>32,677</td>
</tr>
<tr>
<td>Maximum number of columns in an ORDER BY clause</td>
<td>1,012</td>
</tr>
<tr>
<td>Maximum number of prepared statements</td>
<td>storage capacity</td>
</tr>
<tr>
<td>Most declared cursors in a program</td>
<td>storage capacity</td>
</tr>
<tr>
<td>Maximum number of cursors opened at one time</td>
<td>storage capacity</td>
</tr>
<tr>
<td>Maximum number of constraints on a table</td>
<td>storage capacity</td>
</tr>
<tr>
<td>Maximum level of subquery nesting</td>
<td>storage capacity</td>
</tr>
<tr>
<td>Maximum number of subqueries in a single statement</td>
<td>storage capacity</td>
</tr>
<tr>
<td>Maximum number of rows changed in a unit of work</td>
<td>storage capacity</td>
</tr>
<tr>
<td>Most constants in a statement</td>
<td>storage capacity</td>
</tr>
<tr>
<td>Maximum depth of cascaded triggers</td>
<td>16</td>
</tr>
</tbody>
</table>
Notices

IBM may not offer the products, services, or features discussed in this document in all 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/region or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country/region 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 Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product, and use of those Web sites 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 that has been exchanged, should contact:

IBM Corporation
J74/G4
555 Bailey Ave.
San Jose, CA 95161-9023
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.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems, and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements, or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility, or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM’s future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information may contain 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 may contain 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.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:
© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

**Trademarks**

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both, and have been used in at least one of the documents in the DB2 UDB documentation library.

<table>
<thead>
<tr>
<th>Cloudscape</th>
<th>DRDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2</td>
<td>IBM</td>
</tr>
<tr>
<td>DB2 Universal Database</td>
<td>WebSphere</td>
</tr>
</tbody>
</table>

The following terms are trademarks or registered trademarks of other companies and have been used in at least one of the documents in the IBM Cloudscape documentation library:

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.
Index

Special characters
   _ as wildcard within SQL 2
   -- (comment delimiters within SQL) 2
   ? (dynamic parameter)
      in prepared statements 179
      used in conditional expression 180
   * as wildcard in SQL SELECT 1, 72
   % as wildcard in SQL 2
   +, -, *, /, unary + and - expressions 175
   |call (JDBC escape syntax) 226
   |d (JDBC escape syntax) 227
   |o (JDBC escape syntax) 230
   |t (JDBC escape syntax) 231
   |ts (JDBC escape syntax) 232

A
   abs (JDBC scalar function) 229
   ABS or ABSVAL (function) 86
   Adding Columns 9
   Adding Constraints 10
   Aggregates 84
      and data types 84
      definition 84
      requirements for 84
   ALTER TABLE statement 9
   AND boolean operator 177
   ASC
      implied in ORDER BY clause 66
   Attributes to database connection URL 233, 248
   AuthorizationIdentifierName identifier 6
   autogenerated keys 223
   AVG aggregate function 87

B
   Backing indexes
      created automatically to enforce primary key, unique, and foreign key constraints 15, 46
      specifying storage properties for 46
   Batch processing (JDBC) 219, 220
   BETWEEN boolean operator 178
   BIGINT data type 153
   BIGINT function 88
   BLOB data type 154, 215
   Boolean expression 177
   Boolean operators
      part of WHERE clause 177
   bootPassword=key (database connection URL attribute) 234
   Byte strings
      variable-length 173

C
   call (JDBC escape keyword) 226
   CALL (PROCEDURE) 42
   CASE expression 95
   Case sensitivity of keywords and identifiers 1
   CAST expression 175
   CAST function 89
CHAR data type 155
CHAR FOR BIT DATA type 156
CHAR function 91
Character expressions 176
Check constraints 46
Class path
accessing read-only databases from 205
CLOB data type 157, 215
Column defaults 20
Column reference expression 174
Column widths
increasing 10
Column-level constraints 43
used in CREATE TABLE statement 43
column–Name identifier 5
Columns
adding 9
modifying 10
Comment delimiters within SQL 2
Comparison boolean operators 177
Compressing tables 9
Compressing Tables 136
concat (JDBC scalar function) 229
Concatenation function 94
as expression 176
Conditional expression 175
Connecting to a database (using databaseName attribute) 236
Connection pooling 249, 251
Connections
garbage collection of 207
session-severity exceptions closing 207
system-severity exceptions closing 207
Constant expression 174
Constraint clause
referential actions in 46
CONSTRAINT clause 43
correlation–Name identifier 4
Constraints 43
adding 9
column-level 43
compressing 9
differences between column-level and table-level 44
dropping 9
table-level 43
correlation–Name identifier 4
COUNT aggregate function 96
COUNT(*) aggregate function 97
CREATE FUNCTION statement 13
CREATE INDEX statement 15
CREATE PROCEDURE statement 17
CREATE SCHEMA statement 19
CREATE TABLE statement 20
CREATE TRIGGER statement 23
CREATE VIEW statement 27
createFrom=Path (database connection URL attribute) 244
create=true (database connection URL attribute) 235
CURRENT DATE function 98
CURRENT ISOLATION function 100
CURRENT TIME function 102
CURRENT TIMESTAMP function 104
CURRENT_DATE function 99
as expression 176
CURRENT_TIME function 103
as expression 176
CURRENT_TIMESTAMP function 105
expression 176
CURRENT_USER function 106
cursor–Name identifier  6
Cursors  54
    pre-fetching affecting  213
    scrolling insensitive  218

D
d (JDBC escape keyword)  227
Data encryption
    algorithms  239
    providers  238
Data sources  251
    creating a database for  251
    shutting down a database for  251
Data types
    modifying  9, 10
Database connection URL  205
    attributes  206
    settable through Properties in connection call  206
    syntax in embedded mode  205
Database Manager limitations  259
DatabaseMetaData.getProcedureColumns  208
databaseName=nameOfDatabase (database connection URL attribute)  236
Databases
    encrypting  234, 237
    territory  240
dataEncryption=true (database connection URL attribute)  237
DATE data type  159
Date formats
    JDBC escape syntax for  227
DATE function  107
DATE limitations  258
Date/time expressions  176
DAY function  108
DECIMAL data type  160
DECLARE GLOBAL TEMPORARY TABLE statement  50
Defaults  20
    adding  9, 10
    dropping  9, 10
    setting  11
DELETE statement  52
delimited identifiers
    definition  2
Dependencies
    internal tracking of  7
DESC
    optional element of ORDER BY clause  66
Descending indexes  15
Dictionary objects
    definition  2
DISTINCT
    qualifier in AVG function  87
    qualifier in COUNT aggregate function  96
DISTINCT clause
    part of SelectExpression  71
Distributed transaction management  249
DOUBLE data type  161
DOUBLE function  109
DOUBLE PRECISION data type  162
DROP FUNCTION statement  28
DROP INDEX statement  29
DROP PROCEDURE statement  30
DROP SCHEMA statement  31
DROP TABLE statement  32
DROP TRIGGER statement  33
DROP VIEW statement  34
Dropping Constraints  10
Dynamic parameter expression 175
Dynamic parameters 179

E
Encrypting databases 234, 237
Encryption
   algorithms 239
   providers 238
encryptionAlgorithm=algorithm (database connection URL attribute) 239
encryptionProvider=providerName (database connection URL attribute) 238
Escape (JDBC escape keyword) 228
Escape character
   for single- quotation mark 1
Escape character for LIKE clauses 228
Escape syntax (JDBC) 224
EXISTS boolean operator 178
Export
   SYSCS_UTIL.SYSCS_EXPORT_QUERY system procedure 141
   SYSCS_UTIL.SYSCS_EXPORT_TABLE system procedure 140
Expression subquery 70
Expressions
   hierarchy of 176
   in SQL language 174
   precedence of 176

F
FLOAT data type
   as alias for REAL or DOUBLE PRECISION 163
   java.sql type converted to DOUBLE PRECISION when retrieved 215
fn (JDBC escape keyword) 229
FOR UPDATE clause 54
Foreign keys 45
FROM clause 55
FUNCTION
   Create 13
Functions, built-in 83

G
getAsciiStream 213
getBinaryStream 213
getUnicodeStream 213
getXXXStream requests 213
GROUP BY clause 56

H
HAVING clause 57
HOUR function 110

I
Identifier length limitations 257
Identifiers
   definition 2
Identity Column Attributes 20
Identity columns
   and triggers 21
Import
   SYCS_UTIL.SYSCS_IMPORT_DATA system procedure 143
   SYSCS_UTIL.SYSCS_IMPORT_TABLE system procedure 142
IN (? or dynamic) parameters 179
IN boolean operator 178

268 IBM Cloudscape: Reference Manual
In-place updates 54
index–Name identifier 5
Indexes
  automatic creation of for primary key, unique, and foreign key constraints 15, 46
  created in descending order 15
  specifying page size for 15
INNER JOIN expression 58
INSERT statement 60
INTEGER data type 164
INTEGER function 113
Internationalization features
  territory 240
IS NOT NULL boolean operator 177
IS NULL boolean operator 177
ISOLATION 41
Isolation levels
  setting via JDBC 207

J
J2EE 249
Java identifiers
  case sensitivity of within SQL 1
java.sql.Blob 215
java.sql.CallableStatement 211
  and INOUT parameters 212
  and OUT parameters 212
java.sql.Clob 215
java.sql.Connection 207
java.sql.Connection.isReadOnly 207
java.sql.Connection.setReadOnly 207
java.sql.Connection.setTransactionIsolation 207
java.sql.DatabaseMetaData 207
java.sql.Driver 204
java.sql.Driver.getPropertyInfo 206
java.sql.DriverManager.getConnection 204
java.sql.ParameterMetaData 221
java.sql.PreparedStatement 210
java.sql.ResultSet 213
java.sql.ResultSetMetaData 214
java.sql.Savepoint 222
java.sql.SQLException 214
java.sql.SQLWarning 214
java.sql.Statement 209
java.sql.Types
  mapped to SQL types 214
java.transaction.xa package 249
JavaIdentifiers
  definition 2
java.sql package 251
java.sql.ConnectionPoolDataSource 249, 251
java.sql.DataSource 249, 251
java.sql.PooledConnection 249, 251
java.sql.XAConnection 251
java.sql.XADataSource 251
java.sql.XAxxx interfaces 249
java.transaction.xa.XAException 250
java.transaction.xa.XAResource 250
java.transaction.xa.Xid 250
JDBC 2.0
  standard extensions 251
JDBC 3.0
  Cloudscape’s support for 220
JDBC API
  Cloudscape implementation of 203
JDBC driver
  loading 204
JDBC escape syntax 224
JDBC types
  mapped to SQL types 214
jdbc.drivers system property
  using to load driver 204
JNDI 249
JOIN operation 62
JTA API 250

K
Keywords 181
  case insensitivity of 1

L
LCASE or LOWER (function) 115
LEFT OUTER JOIN expression 63
LENGTH function 93, 176
LIKE
  escape character for 178
LIKE boolean operator 178
Limitations 257
  Database Manager 259
  DATE, TIME, and TIMESTAMP 258
  Identifier length 257
  Numeric 257
  String 258
locate (JDBC scalar function) 229
LOCATE function 114
Lock escalation
  by user 64
LOCK TABLE statement 64
logDevice=logDirectoryPath (database connection URL attribute) 241
LONG VARCHAR data type 165
LONG VARCHAR FOR BIT DATA type 166
LONGVARBINARY 215
LONGVARCHAR 215
LTRIM (function) 116

M
Mapping
  of SQL types to java.sql.Types 215
MAX aggregate function 117
MIN aggregate function 118
MINUTE function 119
mod (JDBC scalar function) 229
MOD function 120
MONTH function 121

N
new–table–Name identifier 4
NOT boolean operator 177
NULL
  not a data type 145
NULL expression 174
NULLIF expression 95
NUMERIC data type 167
Numeric data types 146
  implicit conversion 147
  promotion of in expressions 146
Numeric expressions 175
Numeric limitations 257
of (JDBC escape keyword)  230
ON clause
   part of INNER JOIN expression  58
OR boolean operator  177
ORDER BY clause  66
   affecting cursors  66
      needed to guarantee order of results  74
org.apache.derby.authentication.UserAuthenticator  254
org.apache.derby.jdbc.EmbeddedConnectionPoolDataSource  253
org.apache.derby.jdbc.EmbeddedDataSource  253
org.apache.derby.jdbc.EmbeddedDriver  253
org.apache.derby.jdbc.EmbeddedXADatasource  253
org.apache.derby.jdbc.EmbeddedDriver  253
Outer joins  62
   JDBC escape syntax for  230

password=userPassword (database connection URL attribute)  242
Precision
   of decimal arithmetic expressions  147
Prepared statements
   tracking of dependencies  7
   use of dynamic parameters in  179
Primary key constraints  44
Primary keys
   adding  43
   creating  43
Procedure
   Create  17

Quantified comparison boolean operator  179

Read-write VTIs
   and DELETES  52
   and INSERTs  60
REAL data type  168
Referential actions  46
Referential integrity
   via foreign keys  45
REN AME INDEX statement  36
REN AME TABLE statement  37
Reserved words  181
Resource manager
   Cloudscape as in J2EE system  249
restoreFrom=Path (database connection URL attribute)  245
ResultSets
   errors in statement affecting creation of  209
   RIGHT OUTER JOIN expression  69
rollforwardRecovery=Path (database connection URL attribute)  243
RTRIM (function)  122

Savepoints
   methods  223
   releasing  223
   setting and rolling back  222
   setting and rolling back to  222
Savepoints (JDBC)  221, 222
Scalar subquery expression 175
ScalarSubquery 70
Scale
  for decimal arithmetic 147
  of decimal arithmetic expressions 147
schema–Name identifier 3
Schemas
  APP as default 204
  changing default for session 39, 101
  creating 19
  dropping 31
Scroll insensitive cursors (JDBC) 218
Scrolling insensitive cursors 218
SECOND function 123
Select expression 71
SELECT statement 74
SESSION_USER (function) 124
Set functions 84
SET ISOLATION statement 41
SET SCHEMA statement 39, 101
setAsciiStream 210
setBinaryStream 210
setUnicodeStream 210
setXXXStream requests 210
shutdown=true (database connection URL attribute) 246
Simple–column–Name identifier 5
SMALL INTEGER data type 169
SMALLINT function 125
Special characters
  escaping in SQL statements 1
SQL identifiers
  definition 2
SQL-92
  features not supported by Cloudscape 187
  features supported by Cloudscape 187
SQL-92 data types
  supported by Cloudscape 145
SQL92Identifiers
  definition 2
  rules for 2
SQLExceptions
  chaining 201, 214
SQLState values 201
SQLState XJZZZ
  meaning of 203
SQRT (function) 126
sqrt (JDBC scalar function) 229
Statements
  CALL (PROCEDURE) 42
  DROP FUNCTION 28
  DROP PROCEDURE 30
Storage properties
  for indexes backing constraints 46
Stream 210
Streaming columns
  as IN parameters to prepared statement 210
  retrieving from ResultSets using getXXXStream methods 213
String expressions
  with wildcards 176
String limitations 258
Strings
  delimited by single quotation marks within SQL statements 1
SUBSTR (function) 127
substring (JDBC scalar function) 229
Subsubprotocol
  part of database connection URL syntax 205
SUM aggregate function 128
T

t (JDBC escape keyword) 231

Table constraints
    used in CREATE TABLE statement 43

Table subquery expression 175

Table-level constraints 43
table–Name identifier 4

TableExpression 76

Tables
    altering 9
    changing lock granularity for 11
    compressing 9
    dropping 32
    locking 64
    specifying page size for 46
    specifying properties for 46
    Temporary 50

TableSubquery 77
Territories supported 255
territory database 240
territory=ll_CC (database connection URL attribute) 240
TIME data type 170
Time formats
  JDBC escape syntax for 231
  TIME function 129
  TIME limitations 258
TIMESTAMP data type 171
Timestamp formats
  JDBC escape syntax for 232
  TIMESTAMP function 130
  TIMESTAMP limitations 258
TriggerName identifier 6
Triggers
  and database events 23
  and recursion 25
  and the referencing clause 23
  creating 23
  defining when they fire 23
  dropping 33
  order of execution 25
  referencing old and new values 23
  statement versus row triggers 24
  the triggered–SQL–statement 25
  ts (JDBC escape keyword) 232
Type correspondence
  for JDBC and SQL types 214

U
Unicode
  setUnicodeStream 210
Unicode escapes
  support for in SQL statements 1
UNION 67
UNION ALL 67
Unique constraints 44
UPDATE statement 78
UPPER function 131
USER function 132
user=userName (database connection URL attribute) 247

V
VALUES expression 80
VARCHAR data type 172
VARCHAR FOR BIT DATA type 173
VARCHAR function 133
view–Name identifier 4
Views
  creating 27
  dropping 34

W
WHERE clause 81
  SQL extensions to 81
WHERE CURRENT OF clause 82
Wildcards in SQL 2